



Basic Research in Computer Science

**Program Verification with
Monadic Second-Order Logic
&
Languages for
Web Service Development**

Anders Møller

BRICS Dissertation Series

DS-02-4

ISSN 1396-7002

September 2002

Copyright © 2002,

Anders Møller.

**BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK-8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`

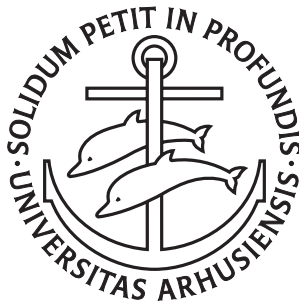
`ftp://ftp.brics.dk`

This document in subdirectory DS/02/4/

Program Verification with
Monadic Second-Order Logic
&
Languages for
Web Service Development

Anders Møller

Ph.D. Dissertation



Department of Computer Science
University of Aarhus
Denmark

Program Verification with
Monadic Second-Order Logic
&
Languages for
Web Service Development

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Anders Møller
June 12, 2002

Abstract

Domain-specific formal languages are an essential part of computer science, combining theory and practice. Such languages are characterized by being tailor-made for specific application domains and thereby providing expressiveness on high abstraction levels and allowing specialized analysis and verification techniques. This dissertation describes two projects, each exploring one particular instance of such languages: monadic second-order logic and its application to program verification, and programming languages for construction of interactive Web services. Both program verification and Web service development are areas of programming language research that have received increased attention during the last years.

We first show how the logic Weak monadic Second-order Logic on Strings and Trees can be implemented efficiently despite an intractable theoretical worst-case complexity. Among several other applications, this implementation forms the basis of a verification technique for imperative programs that perform data-type operations using pointers. To achieve this, the basic logic is extended with layers of language abstractions. Also, a language for expressing data structures and operations along with correctness specifications is designed. Using Hoare logic, programs are split into loop-free fragments which can be encoded in the logic. The technique is described for recursive data types and later extended to the whole class of graph types. As an example application, we verify correctness properties of an implementation of the insert procedure for red-black search trees.

We then show how Web service development can benefit from high-level language support. Existing programming languages for Web services are typically general-purpose languages that provide only low-level primitives for common problems, such as maintaining session state and dynamically producing HTML or XML documents. By introducing explicit language-based mechanisms for those issues, we liberate the Web service programmer from the tedious and error-prone alternatives. Specialized program analyses aid the programmer by verifying at compile time that only valid HTML documents are ever shown to the clients at runtime and that the documents are constructed consistently. In addition, the language design provides support for declarative form-field validation, caching of dynamic documents, concurrency control based on temporal-logic specifications, and syntax-level macros for making additional language extensions. In its newest version, the programming language is designed as an extension of Java. To describe classes of XML documents, we introduce a novel XML schema language aiming to both simplify and generalize existing proposals. All parts are implemented and tested in practice.

Both projects involve design of high-level languages and specialized analysis and verification techniques, supporting the thesis that the domain-specific paradigm can provide a versatile and productive approach to development of formal languages.

Acknowledgments

I sincerely thank Claus Brabrand, Rasmus Pagh, Anna Östlin, Jacob and Louise Elgaard, Anders Sandholm, and Aske Simon Christensen for good friendship and for making DAIMI a great place to be.

I am grateful to Michael Schwartzbach for being a wise and witty mentor. He has been the ideal supervisor during my studies.

I have learned much from Nils Klarlund. I thank him for valuable cooperation and guidance, and for his hospitality the times I visited New Jersey.

I am also indebted to Alex Aiken for allowing me to have an inspiring and exciting stay at UC Berkeley. Also thanks to Jeff Foster, Zhendong Su, and David Gay for their friendliness and insightful discussions.

Special thanks go to my wife Hanne and to my parents for encouragement and giving me the possibility to pursue my goals. My daughter Sara Louise arrived just in time for this section to be written. I appreciate her effort in reminding me that there are much more important things in life than programming languages.

*Anders Møller,
Aarhus, June 12, 2002.*

Contents

Abstract	v
Acknowledgments	vii
I Overview	1
1 Introduction	3
1.1 Structure of the Dissertation	3
1.2 About Domain-Specific Formal Languages	5
2 Program Verification with Monadic Second-Order Logic	7
2.1 The MONA Tool	7
2.1.1 The Automaton–Logic Connection	8
2.1.2 Core WS1S	9
2.1.3 Complexity	10
2.1.4 BDD Representation of Automata	11
2.1.5 Tree Logics and Tree Automata	12
2.1.6 Finite vs. Infinite Structures	12
2.1.7 Restrictions and Three-Valued Logic	13
2.1.8 Other Implementation Tricks	13
2.1.9 Applications	17
2.2 Program Verification	19
2.2.1 Overview	19
2.2.2 Related Work	20
2.2.3 Pointer Assertion Logic and Graph Types	21
2.2.4 Encoding Programs and Properties in MONA Logic	23
3 Languages for Web Service Development	27
3.1 Interactive Web Services	28
3.2 The Session-Centered Approach	31
3.2.1 Script-Centered Languages	31
3.2.2 Page-Centered Languages	32
3.2.3 Session-Centered Languages	33
3.2.4 A Runtime Model with Support for Sessions	35
3.3 Dynamic Construction of Web Pages	36
3.4 Program Analyses	39

3.4.1	Flow Graphs for Jwig Programs	40
3.4.2	Summary Graphs for XML Expressions	40
3.4.3	Analyzing Plug Operations	42
3.4.4	Analyzing Receive Operations	42
3.4.5	Analyzing Show Operations	43
3.5	Declarative Form Field Validation	43
3.5.1	The PowerForms Language	44
3.5.2	PowerForms in Jwig	46
3.6	Other Aspects of Web Service Languages	46
3.6.1	Concurrency Control	46
3.6.2	Security Issues	48
3.6.3	Language Abstractions with Syntax Macros	49
3.7	Schema Languages for XML	49
3.7.1	The Document Structure Description Language	51
3.7.2	Validating Summary Graphs with DSD2	54
4	Conclusion	57
II	Publications	59
5	MONA 1.x: New Techniques for WS1S and WS2S	61
5.1	Introduction	61
5.2	M2L(Str) and WS1S	62
5.3	DAGs for Compilation	64
5.4	Experimental Results	64
5.5	Related and Future Work	65
6	MONA Implementation Secrets	67
6.1	Introduction	67
6.2	The Automaton–Logic Connection	68
6.3	Benchmark Formulas	70
6.4	Implementation Secrets	72
6.4.1	BDD-based Automata Representation	72
6.4.2	Cache-Conscious Data Structures	73
6.4.3	Eager Minimization	74
6.4.4	Guided Tree Automata	75
6.4.5	DAGification	75
6.4.6	Three-Valued Logic and Automata	76
6.4.7	Formula Reductions	77
6.5	Future Developments	79
6.6	Conclusion	80
7	Compile-Time Debugging of C Programs Working on Trees	81
7.1	Introduction	81
7.2	The Language	85
7.2.1	The C Subset	85

7.2.2	Modeling the Store	86
7.2.3	Store Logic	87
7.2.4	Program Annotations and Hoare Triples	88
7.3	Deciding Hoare Triples	90
7.3.1	Weak Monadic Second-Order Logic with Recursive Types . .	90
7.3.2	Encoding Stores and Formulas in WSRT	91
7.3.3	Predicate Transformation	92
7.4	Deciding WSRT	93
7.4.1	The Naive Decision Procedure	93
7.4.2	A Decision Procedure using Guided Tree Automata	94
7.5	Conclusion	95
8	The Pointer Assertion Logic Engine	97
8.1	Introduction	97
8.1.1	A Tiny Example	99
8.1.2	Related Work	100
8.2	Pointer Assertion Logic	101
8.2.1	Store Model	101
8.2.2	Graph Types	101
8.2.3	The Programming Language	103
8.2.4	Program Annotations	104
8.2.5	Semantics of Annotations	106
8.3	Example: Threaded Trees	106
8.4	Hoare Logic Revisited	108
8.5	Deciding Hoare Triples in MONA	109
8.6	Data Abstractions	112
8.7	Implementation and Evaluation	113
8.8	Conclusion	116
9	The <bigwig> Project	117
9.1	Introduction	117
9.1.1	Motivation	118
9.1.2	The <bigwig> Language	120
9.1.3	Overview	122
9.2	Session-Centered Web Services	122
9.2.1	The Script-Centered Approach	122
9.2.2	The Page-Centered Approach	124
9.2.3	The Session-Centered Approach	125
9.2.4	Structure of <bigwig> Services	126
9.2.5	A Session-Based Runtime Model	127
9.3	Dynamic Construction of HTML Pages	129
9.3.1	Analysis of Template Construction and Form Input	131
9.3.2	HTML Validity Analysis	132
9.3.3	Caching of Dynamically Generated HTML	134
9.3.4	Code Gaps and Document Clusters	134
9.4	Form Field Validation	135
9.5	Concurrency Control	138

9.6	Syntax Macros	140
9.7	Other Web Service Aspects	143
9.7.1	HTML Deconstruction	143
9.7.2	Seslets	144
9.7.3	Databases	144
9.7.4	Security	145
9.8	Evaluation	146
9.8.1	Experience with <bigwig>	146
9.8.2	Performance	147
9.9	Conclusion	147
9.9.1	Acknowledgments	148
10	A Runtime System for Interactive Web Services	149
10.1	Introduction	149
10.2	Motivation	150
10.2.1	The Session Concept	150
10.2.2	CGI Scripts and Sequential Session Threads	151
10.2.3	Other CGI Shortcomings	151
10.2.4	Handling Safety Requirements Consistently	152
10.3	Components in the Runtime System	153
10.4	Dynamics of the Runtime System	154
10.4.1	Execution of a Thread	155
10.4.2	Starting up a Session Thread	155
10.4.3	Interaction with the Client	156
10.4.4	Interaction with the Controller	157
10.5	Extending the Runtime System	160
10.6	Related Work	161
10.7	Conclusions and Future Work	162
11	PowerForms: Declarative Client-Side Form Field Validation	165
11.1	Introduction	166
11.1.1	Input Validation	166
11.1.2	Field Interdependencies	167
11.1.3	JavaScript Programming	167
11.1.4	Our Solution: PowerForms	168
11.1.5	Related Work	168
11.2	Validation of Input Formats	169
11.2.1	Syntax	169
11.2.2	Semantics of Regular Expressions	170
11.2.3	Semantics of Format Declarations	171
11.2.4	Examples	172
11.3	Interdependencies of Form Fields	174
11.3.1	Syntax	175
11.3.2	Semantics of Boolean Expressions	175
11.3.3	Semantics of Interdependencies	175
11.3.4	Examples	176
11.4	Applet Results	181

11.5	Translation to JavaScript	181
11.6	Availability	182
11.7	Conclusion	182
12	Language-Based Caching of Dynamically Generated HTML	183
12.1	Introduction	183
12.2	Related Work	185
12.3	Dynamic Documents in <bigwig>	188
12.3.1	Dynamic Document Representation	190
12.4	Client-Side Caching	192
12.4.1	Caching	193
12.4.2	Compact Representation	194
12.4.3	Clustering	195
12.5	Experiments	196
12.6	Future Work	199
12.7	Conclusion	199
13	Static Validation of Dynamically Generated HTML	201
13.1	Introduction	201
13.1.1	Outline	202
13.2	XHTML Documents in <bigwig>	202
13.2.1	XML Templates	204
13.2.2	Programs	205
13.3	Summary Graphs	205
13.4	Gap Track Analysis	207
13.4.1	Lattices	207
13.4.2	Transfer Functions	207
13.4.3	The Analysis	208
13.5	Summary Graph Analysis	208
13.5.1	Lattices	208
13.5.2	Transfer Functions	208
13.5.3	The Analysis	209
13.5.4	The Example Revisited	210
13.6	An Abstract DTD for XHTML	211
13.6.1	Examples for XHTML	213
13.6.2	Exceptions in <bigwig>	214
13.7	Validating Summary Graphs	214
13.8	Experiments	216
13.8.1	Error Diagnostics	217
13.9	Related Work	218
13.10	Extensions and Future Work	218
13.11	Conclusion	219
14	The DSD Schema Language	221
14.1	Introduction	222
14.1.1	Outline	226
14.2	XML Concepts	226

14.3	The DSD Language	226
14.3.1	Element Constraints	227
14.3.2	Attribute Declarations	228
14.3.3	String Types	229
14.3.4	Content Expressions	230
14.3.5	Context Patterns	232
14.3.6	Default Insertion	234
14.3.7	ID Attributes and Points-To Requirements	235
14.3.8	Redefinitions and Evolving DSDs	236
14.3.9	Self-documentation	237
14.3.10	The Meta-DSD	237
14.4	The Book Example	237
14.5	The DSD 1.0 Tool	240
14.6	Industrial Case Study: IVR Systems	241
14.6.1	The IVR Scenario	242
14.6.2	DSDs for Syntax Explanations	242
14.6.3	DSDs for Debugging	243
14.6.4	DSDs for Myriads of Defaults	244
14.6.5	DSDs for Simplifying XPML Processing	246
14.6.6	Summary of DSD Advantages	247
14.7	Related Work	247
14.7.1	XML Schema	248
14.7.2	RELAX NG	251
14.7.3	Other Proposals	252
14.8	Conclusion	253
15	Extending Java for High-Level Web Service Construction	255
15.1	Introduction	255
15.1.1	Sessions and Web Pages	256
15.1.2	Contributions	257
15.1.3	Problems with Existing Approaches	257
15.1.4	Outline	260
15.2	The JWIG Language	260
15.2.1	Program Structure	260
15.2.2	Client Interaction	262
15.2.3	Dynamic Document Construction	263
15.2.4	The JWIG Program Translation Process	267
15.2.5	An Example JWIG Program	268
15.3	Flow Graph Construction	269
15.3.1	Structure of Flow Graphs	269
15.3.2	Semantics of Flow Graphs	270
15.3.3	From JWIG Programs to Flow Graphs	272
15.3.4	Complexity	277
15.3.5	Flow Graph for the Example	278
15.4	Summary Graph Analysis	278
15.4.1	String Analysis	279
15.4.2	Summary Graphs	280

15.4.3	Constructing Summary Graphs	283
15.4.4	Summary Graphs for the Example	285
15.5	Providing Static Guarantees	286
15.5.1	Plug Analysis	286
15.5.2	Receive Analysis	287
15.5.3	Show Analysis	290
15.5.4	The Document Structure Description 2.0 Language	291
15.5.5	Validity Analysis	294
15.6	Implementation and Evaluation	297
15.6.1	Example: The Memory Game	298
15.6.2	Performance	302
15.7	Plans and Ideas for Future Development	305
15.7.1	Language Design	305
15.7.2	Program Analysis	306
15.7.3	Implementation	306
15.8	Conclusion	307
16	Static Analysis for Dynamic XML	309
16.1	Introduction	309
16.2	XML Templates	310
16.3	Summary Graphs	311
16.4	Static Guarantees in JWIG	314
16.5	Analyzing Deconstruction	314
16.6	Regular Expression Types	318
16.7	Conclusion	320
	Bibliography	321

Part I

Overview

Chapter 1

Introduction

1.1 Structure of the Dissertation

This dissertation documents the author's scientific work during his PhD studies at BRICS, Department of Computer Science, University of Aarhus. Part I contains a general overview of the research areas and the author's contributions; Part II contains a collection of co-authored papers. The work has resulted in the following papers, here listed in chronological order of publication or submission:

MONA 1.x: New Techniques for WS1S and WS2S

with Jacob Elgaard and Nils Klarlund, published in *Proc. 10th International Conference on Computer Aided Verification, CAV '98*, LNCS vol. 1427, pp. 516–520, Springer-Verlag, June/July 1998.

A Runtime System for Interactive Web Services

with Claus Brabrand, Anders Sandholm, and Michael I. Schwartzbach, published in *Proc. 8th International World Wide Web Conference, WWW8*, pp. 313–324, Elsevier, May 1999; also in *Computer Networks* Vol. 31 No. 11–16, pp. 1391–1401, Elsevier, May 1999.

Compile-Time Debugging of C Programs Working on Trees

with Jacob Elgaard and Michael I. Schwartzbach, published in *Programming Languages and Systems, Proc. 9th European Symposium on Programming, ESOP '00*, LNCS vol. 1782, pp. 182–194, Springer-Verlag, March/April 2000.

PowerForms: Declarative Client-Side Form Field Validation

with Claus Brabrand, Mikkel Ricky, and Michael I. Schwartzbach, published in *World Wide Web Journal*, Vol. 3, No. 4, pp. 205–214, Kluwer, December 2000.

Document Structure Description 1.0

with Nils Klarlund and Michael I. Schwartzbach, published in *BRICS Notes Series*, NS-00-7, Department of Computer Science, University of Aarhus, December 2000, 40 pp.

MONA 1.4 User Manual

with Nils Klarlund, published in *BRICS Notes Series*, NS-01-1, Department of Computer Science, University of Aarhus, January 2001, 81 pp.

Static Validation of Dynamically Generated HTML

with Claus Brabrand and Michael I. Schwartzbach, published in *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, pp. 38–45, June 2001.

The Pointer Assertion Logic Engine

with Michael I. Schwartzbach, published in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '01*, pp. 221–231, June 2001; also in *SIGPLAN Notices* Vol. 36, No. 5, May 2001.

The XML Revolution — Technologies for the future Web

with Michael I. Schwartzbach, on-line tutorial, published in *BRICS Notes Series*, NS-01-8, Department of Computer Science, University of Aarhus, December 2001, 186 pp.; revision of BRICS NS-00-8. (This tutorial has since its first edition from February 2000 been visited by more than 90,000 people, counted as different IP numbers.)

The <bigwig> Project

with Claus Brabrand and Michael I. Schwartzbach, to appear in *Transactions on Internet Technology*, Vol. 2, No. 2, ACM, May 2002, 33 pp.

MONA Implementation Secrets

with Nils Klarlund and Michael I. Schwartzbach, to appear in *International Journal of Foundations of Computer Science*, World Scientific, 2002; preliminary version published in *Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00*, LNCS vol. 2088, pp. 182–194, Springer-Verlag, July 2000.

The DSD Schema Language

with Nils Klarlund and Michael I. Schwartzbach, in *Automated Software Engineering*, Vol. 9, No. 3, pp. 285–319, Kluwer, 2002; preliminary version published in *Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00*, pp. 101–111, August 2000.

Language-Based Caching of Dynamically Generated HTML

with Claus Brabrand, Steffan Olesen, and Michael I. Schwartzbach, to appear in *World Wide Web Journal*, Kluwer, 2002, 19 pp.

Extending Java for High-Level Web Service Construction

with Aske Simon Christensen and Michael I. Schwartzbach, submitted for journal publication; preliminary version published in *BRICS Report Series*, RS-02-11, Department of Computer Science, University of Aarhus, March 2002, 54 pp.

Interactive Web Services with Java

with Michael I. Schwartzbach, on-line tutorial, published in *BRICS Notes Series*, NS-02-1, Department of Computer Science, University of Aarhus, April 2002, 99 pp.

Static Analysis for Dynamic XML

with Aske Simon Christensen and Michael I. Schwartzbach, submitted for work-

shop presentation; published in *BRICS Report Series*, RS-02-24, Department of Computer Science, University of Aarhus, May 2002, 13 pp.

The papers that have been refereed and presented at workshops or conferences or published in journals and the two currently in submission can be found in Part II. The remaining publications are available from the author's WWW home page at:

<http://www.brics.dk/~amoeller/>

The papers included in Part II appear as published, with the following exceptions: The typography is modified to make the presentation more coherent and the notation consistent; a few errors have been corrected as explained in footnotes; and finally, because the projects have evolved since the first papers were published, additional footnotes are added to clarify the changes.

In connection to the publications listed above, the work has resulted in the following software tools:

MONA – a WS1S/WS2S decision procedure (see Section 2.1)

PALE – an extension of MONA for program verification with Pointer Assertion Logic (see Section 2.2);

<bigwig> – a compiler, program analyzer, and runtime system for the **<bigwig>** programming language (see Section 3.1)

JWIG – a program analyzer and runtime system for the **JWIG** programming language, which is a successor to the **<bigwig>** language (see Section 3.1)

runwig – a module for the Apache Web server, used for both the **<bigwig>** and the **JWIG** runtime systems (see Section 3.2)

dk.brics.automaton – a Java package for performing DFA/NFA and regular expression operations with Unicode alphabet, used in particular for the DSD2 validator in the **JWIG** program analyzer (see Section 3.4) and in the Java version of the PowerForms compiler (see Section 3.5)

DSD/DSD2 – validators for DSD and DSD2 XML schemas (see Section 3.7)

The tools are developed by the author of this dissertation—for the MONA, **<bigwig>**, and **JWIG** tools, in cooperation with the research group. All the tools are freely available under Open Source licences, also from the author's home page.

The reader of the overview chapters in Part I is assumed to be familiar with regular languages, finite-state automata, mathematical logic, pointers in programming languages, and data-flow-based program analysis, as taught in undergraduate computer science courses and to have a basic understanding of HTML, XML, and the World Wide Web.

1.2 About Domain-Specific Formal Languages

Formal languages are ubiquitous in computer science. Programmers typically use a variety of languages in their work, and even though most programming languages from a theoretical point of view have the same expressive power by being Turing complete,

new languages frequently appear. The justification of this myriad of languages lies in their different designs and intended application domains. A recent trend in programming language development is a shift of focus from general-purpose towards domain-specific languages (DSLs) [75, 63], also called very-high-level languages. Instead of aiming for general software development, such a language is designed to address a narrow class of problems, while in return offering compelling advantages:

- it can provide a high level of abstraction with language constructs and customized syntax that closely match the concepts in the problem domain; and
- highly specialized static analyses, verification techniques, and code optimizations are often possible, since not only the programmer but also the compiler has a better chance of understanding what the programs do.

Learning a single general-purpose programming language, such as C or Java, of course may be easier than building a large toolbox of domain-specific languages. However, by mastering a suitable range of such specialized languages, the general benefits can be improved productivity by easier development and maintenance of applications, and often also increased runtime performance.

The remaining chapters of Part I present an overview of monadic second-order logic and its application to program verification and of high-level languages for development of interactive Web services. Throughout this presentation we will see a number of formal languages that are highly specialized to particular domains: the MONA language is a concise notation for regular sets of strings or trees; Pointer Assertion Logic is a language tailor-made for expressing properties of pointer-intensive programs; `<bigwig>` and **JWIG** are programming languages that are designed for the domain of Web services; DynDoc is a sublanguage of both `<bigwig>` and **JWIG** for dynamic construction of Web pages; PowerForms, another sublanguage, is designed for expressing form input requirements; and DSD and DSD2 are languages for specifying the syntax of XML languages, used in particular in **JWIG**. The various projects constituting this dissertation have many differences, but a common theme is the use of the DSL paradigm. Thus, the dissertation can be viewed as a case study of languages and tools for the specific domains of regular languages, Web services, and various sub-domains of these.

The most essential part of this DSL paradigm is to analyze the application domain in order to identify the central concepts that should appear directly as high-level language constructs. We deliberately apply the paradigm in a broader sense than e.g. in [75] since our languages are not necessarily declarative and we do not rely on automated compiler generators. Instead we focus on the design and implementation of the languages and on specialized analysis and verification techniques.

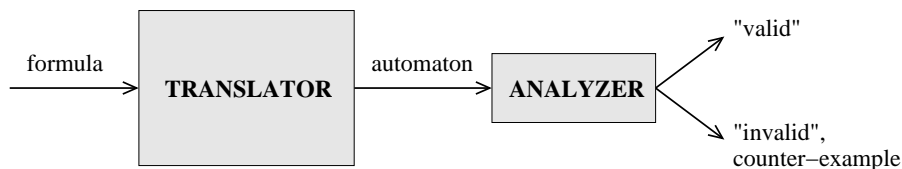
Chapter 2

Program Verification with Monadic Second-Order Logic

The MONA project is an example of theoretical results in basic research evolving into practical tools. It was initiated at BRICS by Klarlund (now at AT&T Labs Research) in 1994, based on Büchi, Elgot, and Trakhtenbrot's classical decidability results for *Weak monadic Second-order theory of 1 Successor*, WS1S, and *Monadic Second-order Logic on finite Strings*, M2L(Str), from the early 1960's [49, 83, 211, 208]. Since 1994, a large number of improvements have been made in the implementation, many people have contributed, and a long range of applications have been published. In this chapter, we describe the theoretical foundation of the MONA tool together with its implementation, and provide an in-depth description of a particular application in the area of program verification.

2.1 The MONA Tool

The MONA tool is a decision procedure for WS1S and a number of related logics. These logics are formal languages for the specific domains of regular sets of finite strings and trees. As input, the tool takes a formula expressed in one of these logics and translates it into a finite-state automaton. By analyzing the automaton, the tool produces as output either the result “valid”, in case the original formula is valid, or the result “invalid” together with a counter-example valuation, in case the formula is not valid:



There are two important practical aspects of this process: 1) Which interesting problems can be expressed in the logics? 2) Can the process be made sufficiently efficient to be practically useful?

Regarding the first aspect, there will obviously be clear theoretical limitations to

expressibility since the involved logics are decidable. Still, the logics are among the most expressive decidable logics that are known, and it is important to investigate the practical usefulness of logics near the edge to undecidability. Also, there is obviously a need for extending the basic languages with convenient “syntactic sugar” to make it more manageable to express problems using the logics. This aspect therefore also involves language design for these highly specialized problem domains.

Regarding the second aspect, one really must be an optimist in order to implement these decision procedures: As explained below, the theoretical worst-case complexity is non-elementary, that is, the time and space requirements are not bounded by any constant-size stack of exponentials in the size of the formulas. This perhaps explains the many years that passed between the discovery of the automaton–logic connection in the 1960’s and the first implementations in the 1990’s. However, it turns out that for many applications, the theoretical bound is not encountered in practice. Experience from the MONA project shows that even for non-elementary decision procedures, it can pay off to develop new representations of formulas and automata, even if they only decrease the resource requirements by perhaps constant factors in average cases.

The WWW home page of the MONA project is located at:

<http://www.brics.dk/mona/>

The tool can be downloaded freely, including the full source code and the MONA User Manual [131]. Since June 1998, there have been around 600 registered downloads and 250 people currently subscribe to the “MONA News” mailing list.

2.1.1 The Automaton–Logic Connection

It has been known for more than forty years that the class of regular languages is linked to decidability questions in formal logics. In particular, the logic WS1S is decidable through the *automaton–logic* connection: the set of satisfying interpretations of a subformula can be represented by a finite-state automaton. This is one of Büchi, Elgot, and Trakhtenbrot’s classical results. The automaton for a formula is calculated inductively: logical connectives correspond to simple automata-theoretic operations such as product and subset construction, and atomic formulas correspond to small basic automata. From the resulting automaton, interesting properties of the original formula, such as validity or counter-examples, can easily be deduced. The MONA tool is an implementation of this decision procedure, where automata are represented using BDDs [47]. In addition, MONA also implements the decision procedure for the logic WS2S (Weak monadic Second-order theory of 2 Successors), a generalization of WS1S.

There are a number of tools closely resembling MONA. Independently of the MONA project, the first implementation of automata represented with BDDs was that of Gupta and Fischer from 1993 [102]. However, their application was representation of *linearly inductive boolean functions* instead of the automaton–logic connection. MOSEL (see <http://sunshine.cs.uni-dortmund.de/projects/mosel/>) implements the automata-based decision procedure for the logic M2L(Str), using BDDs as MONA. In [123], MOSEL is described and compared with MONA 0.2, which provided inspiration for the MOSEL project. Apparently, there have been only few applications of MOSEL. AMORE [153] (see <http://www.informatik.uni-kiel.de/inf/>

Thomas/amore.html) is a library of automata-theoretic algorithms, resembling those used in MONA. AMORE also provides functionality for regular expressions and monoids, but is not tied to the automaton–logic connection. Glenn and Gasarch [98] have in 1997—apparently independently of MONA and MOSEL—implemented a decision procedure for WS1S, basically as the one in MONA, but without using BDDs or other sophisticated techniques. The SHASTA tool from 1998 is based upon the same ideas as MONA. It is used as an engine for Presburger Arithmetic [198]. The Timbuk [94] and RX [215] tools both implement tree automata operations and are used for analyzing term rewriting systems, without the direct relation to logics.

2.1.2 Core WS1S

Being a variation of first-order logic, WS1S is a formalism with quantifiers and boolean connectives. First-order terms denote natural numbers, which can be compared and subjected to addition with constants. In addition, WS1S also allows second-order terms, which are interpreted as finite sets of numbers. The actual MONA syntax is a rich notation with constructs such as set constants, predicates and macros, modulo operations, and let-bindings. If all such syntactic sugar is peeled off, the formulas are “flattened” (so that there are no nested terms), and first-order terms are encoded as second-order terms, the logic reduces to a simple core language:

$$\begin{array}{lll} \varphi ::= & \sim \varphi' & | \quad \varphi' \& \varphi'' & | \quad \text{ex2 } X_i : \varphi' \\ & X_i \text{ sub } X_j & | \quad X_i = X_j \setminus X_k & | \quad X_i = X_j + 1 \end{array}$$

where X ranges over a set of second-order variables.

Given a fixed main formula φ_0 , we define its semantics inductively relative to a finite string w over the alphabet \mathbb{B}^k , where $\mathbb{B} = \{0, 1\}$ and k is the number of variables in φ_0 . We assume every variable of φ_0 is assigned a unique index in the range $1, 2, \dots, k$, and that X_i denotes the variable with index i . The projection of a string w onto the i 'th component is called the X_i track of w . The string w determines an interpretation $w(X_i)$ of X_i defined as the finite set $\{j \mid \text{the } j\text{th bit in the } X_i \text{ track is } 1\}$.

The semantics of a formula φ in the core language can now be defined inductively relative to an interpretation w . We use the notation $w \models \varphi$ (which is read: w satisfies φ) if the interpretation defined by w makes φ true:

$$\begin{array}{ll} w \models \sim \varphi' & \text{iff } w \not\models \varphi' \\ w \models \varphi' \& \varphi'' & \text{iff } w \models \varphi' \text{ and } w \models \varphi'' \\ w \models \text{ex2 } X_i : \varphi' & \text{iff } \exists \text{ finite } M \subseteq \mathbb{N} : w[X_i \mapsto M] \models \varphi' \\ w \models X_i \text{ sub } X_j & \text{iff } w(X_i) \subseteq w(X_j) \\ w \models X_i = X_j \setminus X_k & \text{iff } w(X_i) = w(X_j) \setminus w(X_k) \\ w \models X_i = X_j + 1 & \text{iff } w(X_i) = \{j + 1 \mid j \in w(X_j)\} \end{array}$$

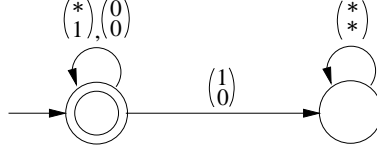
The notation $w[X_i \mapsto M]$ is used for the shortest string that interprets all variables X_j where $j \neq i$ as w does, but interprets X_i as M .

The language $L(\varphi)$ of a formula φ can be defined as the set of satisfying strings: $L(\varphi) = \{w \mid w \models \varphi\}$. By induction in the formula, we can now construct a minimal deterministic finite-state automaton (DFA) A such that $L(A) = L(\varphi)$, where $L(A)$ is the language recognized by A .

For the atomic formulas, we show just one example: the automaton for the formula $\varphi = X_i \text{ sub } X_j$ in the case where $i = 1$ and $j = 2$. The automaton must recognize the language

$$L(X_1 \text{ sub } X_2) = \{w \in (\mathbb{B}^k)^* \mid \text{for all letters in } w: \text{ if the first component is 1, then so is the second } \}$$

Such an automaton is (where $*$ denotes “either 0 or 1”):



The other atomic formulas are treated similarly. The composite formulas are translated as follows:

$\varphi = \sim \varphi'$ Negation of a formula corresponds to automaton complementation. In MONA, this is implemented trivially by flipping accepting and rejecting states.

$\varphi = \varphi' \& \varphi''$ Conjunction corresponds to language intersection. In MONA, this is implemented with a standard automaton product construction generating only the reachable product states. The resulting automaton is minimized.

$\varphi = \text{ex2 } X_i : \varphi'$ Existential quantification corresponds to a simple quotient operation followed by a projection operation. The quotient operation takes care of the problem that the only strings satisfying φ' may be longer than those satisfying $\text{ex2 } X_i : \varphi'$. The projection operation removes the track belonging to X_i , resulting in a nondeterministic automaton, which is subsequently determinized using the subset construction operation, and finally minimized.

This presentation is a simplified version of the procedure actually used in MONA. For more details, see the MONA User Manual [131].

When the minimal automaton A_0 corresponding to φ_0 has been constructed, validity of φ_0 can be checked simply by observing whether A_0 is the one-state automaton accepting everything. If φ_0 is not valid, a (minimal) counter-example can be constructed by finding a (minimal) path in A_0 from the initial state to a non-accepting state.

2.1.3 Complexity

Notice that the most significant source of complexity in this decision procedure is the quantifiers, or more precisely, the automaton determinization. Each quantifier can cause an exponential blow-up in the number of automaton states. In worst case, if n is the length of φ , then the number of states in A is

$$2^{2^{\dots^{2^{c \cdot n}}}} \} c \cdot n$$

for some constant c , although constructing such an example is not trivial. In other words, this decision procedure has a non-elementary complexity. Furthermore, we cannot hope for a fundamentally better decision procedure; in 1972, Meyer showed that this is in fact the lower bound for the WS1S decision problem [156].

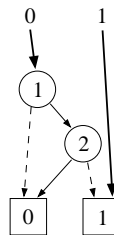
To make matters even worse (and the challenge the more interesting), the implementation also has to deal with automata with huge alphabets. As mentioned, if φ_0 has k free variables, the alphabet is \mathbb{B}^k . Standard automaton packages cannot handle alphabets of that size for typical values of k .

Nevertheless, the non-elementary succinctness of the logic and the exponential alphabet representation can also be seen from an optimistic point of view. If one wants to describe a particular regular set, then a WS1S formula may be non-elementarily more succinct than a regular expression or a transition table. The MONA project has indeed shown that there is a productive niche between the automata that are simple enough to be manually constructed and those that are so complex that no known decision procedure can manage them.

2.1.4 BDD Representation of Automata

The very first attempt to implement the decision procedure used a representation based on conjunctive normal forms. That was indeed not a success. The first version of the MONA tool that was actually useful, was the experimental ML-version from 1995, programmed by Sandholm and Gulmann [105]. The reason for the success was the novel representation of automata based on (reduced and ordered) BDDs (Binary Decision Diagrams) [47, 48]. This representation made it possible to perform verification of parameterized distributed systems using MONA.

A BDD is a graph representing a boolean function. The BDD representation has some extremely convenient properties, such as compactness and canonicity, and it allows efficient manipulation. BDDs have successfully been used in a long range of verification techniques, originally in [51]. In MONA, a special form of BDDs, called *shared multi-terminal* BDDs, or SMBDDs are used. As an example, the transition function of the tiny automaton shown in Section 2.1.2 is represented in MONA as the following SMBDD:



The roots and the leaves represent the states. Each root has an edge to the node representing its alphabet part of the transition function. For the other edges, dashed represents 0 and solid represents 1. As an example, from state 0, the transition labeled $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ leads to state 1. In this way, states are still represented explicitly, but the transitions are represented symbolically.

For many applications, this representation solves the problem of the large alpha-

bets. In addition, the (SM)BDD-based technique allows some specialized algorithms to be applied (see [137, 127]). The resulting effect has been investigated in [134].

Although the problem with the worst-case complexity still exists, removing the bottleneck of the huge alphabets turned out to be the breakthrough needed to make many interesting applications run. It was a surprising result that decision procedures with non-elementary complexity could be of any use at all [18]. Formulas of hundreds of kilobytes have successfully been processed by the MONA tool [134, 136].

2.1.5 Tree Logics and Tree Automata

WS2S, Weak monadic Second-order theory of 2 Successors, is the generalization of WS1S from linear to binary-tree shaped structures [205, 76, 208]. Seen at the “core language” level, WS2S is obtained from WS1S by replacing the single successor predicate by two successor predicates, for *left* and *right* successor respectively. This logic is also decidable by the automaton–logic connection, but using tree automata instead of string automata. The MONA tool also implements this decision procedure.

Moving from string to tree automata introduces an extra source of complexity: the transition tables are now three dimensional as opposed to two dimensional. MONA uses a technique called *Guided Tree Automata* (GTA), which factorizes state spaces to split big tree automata into smaller ones. The basic idea, which may result in exponential savings, is explained in [23] and in [131]. To exploit this feature, the MONA programmer must manually specify a *guide*, which is a top-down tree automaton that assigns state spaces to the tree nodes.

2.1.6 Finite vs. Infinite Structures

There is a subtle difference between WS1S, the logic now used in MONA, and M2L(Str), the logic used until 1996 in the early experimental versions [208, 24, 81]. In WS1S, formulas are interpreted over *infinite string* models (but quantification is restricted to finite sets only). In M2L(Str), formulas are instead interpreted over *finite string* models. That is, the universe is not the whole set of naturals \mathbb{N} , but a bounded subset $\{0, \dots, n-1\}$, where n is defined by the length of the string. The difference between WS2S and M2L(Tree) is similar. The decision procedure for M2L(Str) is almost the same as for WS1S, only slightly simpler: the quotient operation (before the projection) is just omitted. From the language point of view, M2L(Str) corresponds exactly to the regular languages (all formulas correspond to automata *and* vice versa), and WS1S corresponds to those regular languages that are closed under concatenation by 0's. A DFA can be encoded as a M2L(Str) formula by a simple linear-time algorithm.

These properties make M2L(Str) preferable for some applications, for example [18, 194]. However, the fact that not all positions have a successor often makes M2L(Str) rather unnatural to use. Being closer tied to arithmetic, the WS1S semantics is easier to understand. Also, for instance Presburger Arithmetic can easily be encoded in WS1S whereas there is no obvious encoding in M2L(Str).

The logics WS1S and WS2S are closely related to the logics S1S and S2S which are obtained by interpreting with respect to infinite strings and trees instead of finite ones, and equivalently correspond to the regular languages over infinite structures. These logics are also theoretically decidable by generalizing to automata on infinite

strings [50] or trees [183]. However, the Myhill-Nerode theorem does not extend to these ω -automata [125]. This means that such automata do not have effective normal-forms providing a minimization property. As we show in [134], minimization of intermediate automata is crucial in practice.

In a wider perspective, the well-studied temporal logics LTL and CTL* are straightforwardly expressible in S1S and S2S, respectively. WS2S is strictly less expressive than S2S, while, surprisingly, WS1S and S1S are expressively equivalent [208]. However, the latter property is only theoretically useful since no tractable translation is known.

2.1.7 Restrictions and Three-Valued Logic

Using a theory of restrictions and three-valued logic, it is in [128] shown that M2L(Str) efficiently can be emulated in WS1S—provided that an extra construct `allpos(X)` is added to the logic, breaking the closure property mentioned in the previous section.

The basic idea is to add a special second-order variable $\$$ always having the value $\{0, \dots, n-1\}$ for some n , and then *restrict* all values to be subsets of $\$$ using the `allpos` construct. This can in practice be done by associating a *restriction* formula, $restr(X)$ to each variable X , and conjoining $restr(X)$ to formulas using X . Doing this naively can however cause exponential state-space explosions on intermediate automata. To make the emulation efficient, a third truth-value, *don't-care*, is added to the logic and the notion of automata is generalized accordingly. Don't-care states act as reject states, except that they are invariant to negation. The idea is now to only conjoin restriction formulas to *atomic* formulas and convert reject states in the automata that correspond to restriction formulas into don't-care states, such that non-acceptance of restrictions is preserved properly. Although being invented for the emulation of M2L(Str), this restriction technique can presumably be applied in other situations to combat the state-space explosion problem.

By the arguments above, WS1S is preferable to M2L(Str). However, in [10], it is shown that the *bounded model construction* problem has non-elementary complexity for WS1S but exponential complexity for M2L(Str). If one wants a single model instead of a description of all models of a given formula, M2L(Str) may be preferable.

2.1.8 Other Implementation Tricks

There has been no single silver bullet during the development of MONA. Rather, several separate techniques have been discovered, each contributing with some improvement. The BDD, GTA, and restriction techniques have been mentioned in the previous sections. In this section, three other of the most significant techniques are described: DAG representation, formula reduction, and BDD ordering heuristics. We also mention a number of other features for making the MONA tool more practically useful. While the BDD and GTA techniques compared to more naive approaches often provide indispensable exponential savings, the DAG representation and formula reduction typically each yields a factor 2–4 speed-up, in some cases much more. These results together with other implementation choices are described in [134].

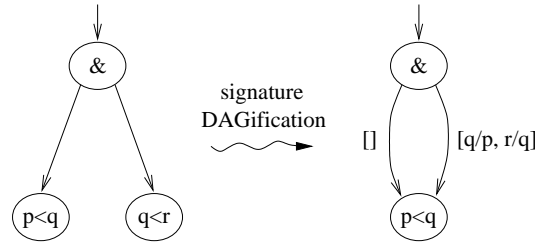
DAG Representation of Formulas

Internally, MONA is divided into a front-end and a back-end. The front-end parses the input and builds a data structure representing the automata-theoretic operations that will calculate the resulting automaton. The back-end then inductively carries out these operations.

The generated data structure is often seen to contain many common subformulas. This is particularly true when they are compared relative to *signature equivalence*, which holds for two formulas ϕ and ϕ' if there is an order-preserving renaming of the variables in ϕ (increasing with respect to the indices of the variables) such that the representations of ϕ and ϕ' become identical.

A property of the BDD representation is that the automata corresponding to signature-equivalent trees are isomorphic in the sense that only the node indices differ. This means that intermediate results can be reused by simple exchanges of node indices. For this reason, MONA represents the formulas in a DAG (Directed Acyclic Graph), not a tree. The DAG is conceptually constructed from the tree using a bottom-up collapsing process, based on the signature equivalence relation, as described in [81].

As an example, consider the formula $\exists x_1 q: p < q \ \& \ q < r$ where the variables p , q , and r have the indices 1, 2, and 3, respectively. The automata for the subformulas $p < q$ and $q < r$ are isomorphic, so their tree nodes are collapsed. The edges of the resulting DAG are labeled with the renaming information:



As shown in [134], this DAGification trick provides a substantial gain in efficiency.

Formula Reductions

Formula reduction is a means of “optimizing” the formulas in the DAG before translating them into automata. The reductions are based on a syntactic analysis that attempts to identify valid subformulas and equivalences among subformulas.

MONA performs three kinds of formula reductions: 1) simple equality and boolean reductions, 2) special quantifier reductions, and 3) special conjunction reductions. The first kind can be described by simple rewrite rules (only some typical ones are shown):

$$\begin{array}{ll}
 X_i = X_i & \rightsquigarrow \text{true} \\
 \text{true} \ \& \ \phi & \rightsquigarrow \phi \\
 \text{false} \ \& \ \phi & \rightsquigarrow \text{false} \\
 \phi \ \& \ \phi & \rightsquigarrow \phi \\
 \sim \sim \phi & \rightsquigarrow \phi \\
 \sim \text{false} & \rightsquigarrow \text{true}
 \end{array}$$

These rewrite steps are guaranteed to reduce complexity, but will not cause significant improvements in running time, since they all either deal with constant size automata

or rarely apply in realistic situations. Nevertheless, they are extremely cheap, and they may yield small improvements, in particular on machine-generated MONA code.

The second kind of reduction can potentially cause tremendous improvements. The non-elementary complexity of the decision procedure is caused by the automaton projection operations, which stem from quantifiers. The accompanying determinization construction may cause an exponential blow-up in automaton size. The basic idea is to apply a rewrite step resembling *let* reduction, which removes quantifiers:

$$\text{ex2 } X_i : \varphi \rightsquigarrow \varphi[T/X_i] \text{ provided that } \varphi \Rightarrow X_i = T \text{ is valid, and } T \text{ is some term satisfying } FV(T) \subseteq FV(\varphi)$$

where $FV(\cdot)$ denotes the set of free variables. For several reasons, this is not the way to proceed in practice. First of all, finding terms T satisfying the side condition can be an expensive task, in worst case non-elementary. Second, the translation into automata requires the formulas to be “flattened” by introduction of quantifiers such that there are no nested terms. So, if the substitution $\varphi[T/X_i]$ generates nested terms, then the removed quantifier is recreated by the translation. Third, when the rewrite rule applies in practice, φ usually has a particular structure as reflected in the following more restrictive rewrite rule chosen in MONA:

$$\text{ex2 } X_i : \varphi \rightsquigarrow \varphi[X_j/X_i] \text{ provided that } \varphi \equiv \dots \& X_i = X_j \& \dots \text{ and } X_j \text{ is some variable other than } X_i$$

In contrast to equality and boolean reductions, this rule is not guaranteed to improve performance since substitutions may cause the DAG reuse degree to decrease, but this is rarely a problem in practice.

The third kind of reductions applies to conjunctions, of which there are two special sources. One is the formula flattening just mentioned; the other is the formula restriction technique mentioned in Section 2.1.6. Both typically introduce many new conjunctions. Studies of a graphical representation of the formula DAGs (see Section 2.1.8) revealed that many of these new conjunctions are redundant. A typical rewrite rule addressing such redundant conjunctions is the following:

$$\varphi_1 \& \varphi_2 \rightsquigarrow \varphi_1 \text{ provided that } \text{unrestr}(\varphi_2) \subseteq \text{unrestr}(\varphi_1) \cup \text{restr}(\varphi_1) \text{ and } \text{restr}(\varphi_2) \subseteq \text{restr}(\varphi_1)$$

Here $\text{unrestr}(\varphi)$ is the set of unrestricted conjuncts in φ , and $\text{restr}(\varphi)$ is the set of restricted conjuncts in φ . This reduction states that it is sufficient to assert φ_1 when $\varphi_1 \& \varphi_2$ was originally asserted in situations where the unrestricted conjuncts of φ_2 are already conjuncts of φ_1 —whether restricted or not—and the restricted conjuncts of φ_2 are unrestricted conjuncts of φ_1 . It is not sufficient that they be restricted conjuncts of φ_1 , since the restrictions may not be the same in φ_1 .

A general benefit from formula reductions is that tools generating MONA formulas from other formalisms may generate more naive and voluminous output while leaving optimizations to MONA. In particular, tools may use existential quantifiers to bind terms to fresh variables, knowing that MONA will take care of the required optimization.

BDD Ordering Heuristics

It is well known for other applications that BDDs often provide compact representations of boolean functions, but in some cases converge towards naive table-based representations yielding exponential blow-ups [48]. The representation depends on the ordering of variables. The optimal ordering can give an exponential improvement in space requirements, but the problem of finding such orderings is NP complete [204]. Numerous publications suggest heuristic approaches for finding close to optimal orderings for various applications, typically based on circuit layouts.

By default, MONA chooses a variable ordering by the order of declaration of the variables in the formula. This is often sufficient, but in some cases far from optimal. Previous versions of MONA allowed the user to manually specify the variable orderings. However, that just pushed the problem to the user who rarely has the required detailed knowledge of the intricate connections between the formulas being constructed and the effect of the BDD variable orderings.

Recently, we have experimented with heuristic techniques based on the formula structure. Preliminary results show promising improvements in memory requirements and running times.

Other Tool Features

In order for a tool such as MONA to be useful in practice, many smaller features have been implemented, often by request by the actual users:

Graphical representations Using the AT&T graphviz tool, MONA can generate illustrations of the formula DAG and the automata. The latter are shown either traditionally (as on page 10), or as (SM)BDDs (as on page 11). This is particularly useful when MONA is used for teaching about automata and logic.

Separate compilation MONA allows its input to be split into several files. Automata can be cached and reused between executions, provided that the files they originate from are not modified. This can speed up the execution for formulas defined using “predicate libraries” located in separate files.

Import and export Some applications, such as [194, 113], do not use MONA as a decision procedure, but rather as an automaton-construction tool. To support this, MONA can import and export automata in a format readable by a small C library, usable for instance as concurrency controllers in Web services or LEGO robots.

Support for easy encoding of other logics MONA has been used as a foundation for deciding other logics, such as Presburger Arithmetic [198], Duration Calculus [180], and WSRT (see Section 2.1.9). This has been supported by adding special constructs. A simple example is `pconst(n)` that creates an automaton recognizing the binary encoding of a number n , used for deciding Presburger Arithmetic.

Prefix closing When WS1S or M2L(Str) is used as a temporal logic, it is usually required that languages are prefix-closed. An automaton prefix-closing operation has been made available for applications synthesizing reactive-system controllers [194, 113].

Inherited acceptance analysis Primarily for use by the YakYak tool [69], an operation analyzing *inherited acceptance* has been implemented. In GTAs, only the root state space has a notion of acceptance, but in YakYak, it is crucial to know whether the property of being in some particular state in some state space is guaranteed to imply acceptance of the whole tree.

These features are all documented in more detail in [131].

2.1.9 Applications

A substantial number of techniques and applications of MONA have been published. Also, MONA has successfully been integrated into or used as foundation of a number of other tools.

In an application perspective, arithmetic and logic is useful because interesting systems and properties can be encoded. A common observation is that WS1S and M2L(Str) can be viewed as a generalization of quantified propositional logic, adding a single “unbounded dimension” orthogonal to the dimension bounded by the number of variables. This unboundedness can generally be used in two ways:

- to model *parameterized systems* and verify a whole family of finite systems at once; or
- to model *discrete time* and verify safety properties or synthesize controllers.

Similarly, the tree logics WS2S and M2L(Tree) can be used to model tree-shaped systems or branching time instead of linear time. Another kind of application is to reduce other logics to the MONA logics to obtain other decision procedures via the MONA tool.

Hardware verification One of the first MONA applications was hardware verification. In [18], the ideas of modeling parameterized systems or discrete time were introduced. In [11] this verification technique is further described and generalized to trees. Many of the applications mentioned below also build on these ideas. In [152], MONA is used for verification of a class of hardware circuits called *systolic arrays*. In [17], it is described how WS1S and MONA can be used to reason about hardware circuits specified in the BLIF description language.

Controller synthesis As mentioned, M2L(Str) can be viewed as a temporal logic, that is, as a logic modeling the occurrence of events over time. To synthesize run-time controllers, MONA turns safety requirements of Web services [194] and LEGO robots [113] into automata, which may act as programs. Whenever a process (a Web session or a LEGO robot) wishes to pass certain “checkpoints”, it is checked that doing so is allowed by the automata, in the sense that reject states are not entered. To keep the automata small, the safety constraints are translated into separate automata, executed as in an implicit product construction.

FIDO FIDO [140] is high-level language built on top of MONA. Its intention is to automatically take care of all the low-level bit-encoding usually required whenever something is encoded directly in MONA logic. The FIDO language is based on *recursive data types over finite domains*, but also adds other programming language concepts, such as subtyping, unification, and coercion. Not exploiting the Guided Tree Automata in MONA, FIDO has to some extent been obsoleted by the WSRT logic, which is described in Section 2.2.

LISA The LISA language [12] was developed in parallel with FIDO. It contains many of the same features as FIDO, but instead of recursive data types, it is based on the more general *feature logics*.

Trace abstractions In [136, 135], FIDO is used to perform behavioral reasoning about distributed reactive systems based on *trace abstractions*. M2L(Str) is used as a temporal logic to address the “Broy-Lamport challenge” about modeling and verification of a memory server specification. This work provided the motivation for the development of FIDO.

Computational linguistics An application of MONA for *linguistic processing and theory verification* using WS2S is described in [164].

Protocol verification MONA has been used for various kinds of protocol verification. In [105], a variant of the Dining Philosophers protocol is verified, and in [200], the Sliding Window communication protocol is modeled using I/O automata and then translated to WS1S and verified. In [179], MONA is used for verification of telephone services.

DCVALID DCVALID [180, 181] is a tool for checking validity of Quantified Discrete-time Duration Calculus formulas based on MONA. It has been used to check properties of SMV, ESTEREL, and SPIN systems.

YakYak YakYak [69] is an extension of the Yacc parser generator. Side constraints expressed in a first-order parse tree logic are translated into Guided Tree Automata using MONA. During the bottom-up Yacc parsing, the parse tree is run on these automata yielding evaluation of the side constraints.

Software engineering In [130], it is shown that many software design architecture descriptions are expressible in M2L(Tree). Using FIDO, parse-tree constraints are expressed and compiled to automata. This project was a precursor of YakYak and did not combine constraint checking with parsing or use Guided Tree Automata.

FMona FMona [29, 28] is a high-level extension of MONA adding e.g. *enumeration types*, *record types*, and *higher-order macros* to the MONA syntax. It has been used to express parameterized transition systems, abstraction relations, synthesis of finite abstractions, and validation of safety properties.

STTools MONA has been used for M2L(Str)-based model checking of programs in the Synchronized Transitions language [187].

PEN PEN [175] is a tool for verifying distributed programs parameterized by the number of processes. The systems are modeled by transducer automata, and properties of configurations are represented by normal automata. By performing transitive closure of the transducer (see [121]) and using an acceleration technique, reachability properties can be verified. The implementation is based on the DFA part of MONA.

PAX PAX [19] is yet another tool for verifying parameterized systems using MONA. The main contribution of PAX is a heuristic-based technique for abstracting parameterized systems in WS1S into finite-state systems for model checking.

PVS MONA has been integrated into the PVS theorem prover [178]. Properties expressible within WS1S can then be verified by PVS without user interaction.

ISABELLE The combination of WS1S and higher-order logic has been investigated using MONA as a WS1S oracle in the ISABELLE system [16].

CACHET MONA has been used to reason about arithmetic and boolean operations in the CACHET system for incremental computation [150].

Program verification The applications [120, 82, 80, 161] are described in more detail in the following section.

Furthermore, the GTA part of MONA is currently being used as basis for an implementation of a decision procedure for an extension of WS1S [124].

2.2 Program Verification

It is notoriously difficult to reason about programs that use pointers, for instance data-type implementations. Many intricate errors can arise, both in the form of memory errors, such as null pointer dereferences, memory leaks, or dangling references, and of programs failing to satisfy more specialized correctness properties, such as maintaining a data-type invariant.

Our verification technique, based on MONA, aims for safety-critical data-type implementations. It requires user guidance in the form of invariants, but, in return, it allows complex properties to be expressed and verified.

2.2.1 Overview

In [120], the MONA logic was used to encode the effect of executing loop-free code with pointer operations on linear heap-based data structures. The approach is based on [139], which introduced a variation of predicate transformation called *transduction*, and is generalized to code containing loops using invariants as in classical Hoare logic [106].

The idea in the transduction technique is to let a family of monadic second-order logic predicates encode a set of heaps. Each primitive step in the loop-free code is simulated by updating the predicates accordingly. As a result, for each program point, there is a predicate family describing how the heap may look at that point. Properties of the heap, such as pre- and post-conditions, can then be encoded using the predicate families and verified using the MONA tool.

We generalize this technique to tree-shaped data structures, or more precisely, *recursive data types*, in [82, 80]. The generalization is conceptually straightforward but nontrivial in practice. As shown in [82], a naive approach cannot exploit the guide in the Guided Tree Automata and hence state-space explosions are inevitable. To overcome this problem, a new logic WSRT, *Weak monadic Second-order logic with Recursive Types*, is introduced. This logic is reminiscent of FIDO and LISA, but is simpler. WSRT permits a more efficient decision procedure: using a technique called *shape encoding*, introduced in [69], a clever guide can automatically be deduced from the recursive types. The transduction method is then performed using WSRT instead of WS2S or M2L(Tree), making the encoding both simpler and faster.

The decision procedure for WSRT is implemented directly in the MONA tool. This means that as a byproduct of the development of this program verification technique, WSRT is now available for other purposes, for instance where WS2S so far has been used. The newest version of the YakYak tool thus uses the WSRT part of MONA.

In [161] we take one step further by showing how the verification approach can be extended to the whole class of data types known as *graph types* [138]. We define a formal language, *Pointer Assertion Logic* (PAL), for expressing properties of the heap structure. This language is essentially a monadic second-order logic in which the universe of discourse contains records, pointers, and booleans. To express data structures and operations we define a specialized programming language for maximizing the potential of the transduction technique. In this language, data structures are declared by graph types, and the only possible values are pointers and booleans. We implement the approach in a tool called PALE, the Pointer Assertion Logic Engine, and experimentally show that the technique works in practice on realistic examples, in spite of the worst case complexity of the logic.

The remainder of this chapter focuses on the approach based on Pointer Assertion Logic. This approach does not apply shape encoding or use the WSRT logic; for further description of those topics, see [82].

2.2.2 Related Work

There are numerous other approaches for modeling data structure operations and verifying correctness properties. These approaches range from approximative code simulators through program analyses to theorem proving. In the light-weight end of the spectrum, only simple properties can be expressed and checked, but often entirely automatically and on large programs. There, the focus is on finding as many bugs as possible, not necessarily all of them. The heavy-weight techniques may require user guidance and generally only work for relatively small programs, but in return, they allow complex properties to be verified. With these techniques, the focus is on guaranteeing correctness, and no bugs should be missed.

General theorem provers, such as HOL [26], may consider the full behavior of

programs but are often slow and not fully automated. Tools such as ESC [74] and LCLint [85] consider memory errors among other undesirable behaviors but usually ignore data structure invariants or only support a few predefined properties. Also, they trade soundness or completeness for efficiency and hence may flag false errors or miss actual errors.

Model checkers such as Bebop [13] and Bandera [65] abstract away the heap and only verify properties of control flow. The JPF [103] model checker verifies simple assertions for a subset of Java, but does not consider structural invariants.

The constraint solver Alloy has been used to verify properties about bounded initial segments of computation sequences [118]. While this is not a complete decision procedure, even for straight-line code, it finds many errors and can produce counterexamples. With this technique, data structure invariants are expressed in first-order logic with transitive closure. However, since it assumes computation bounds, absence of error reports does not imply a guarantee of correctness, and the technique does not appear to scale well.

The symbolic executor PREFIX [52] simulates unannotated code through possible executions paths and detects a large class of errors. Again, this is not a complete or sound decision procedure, and data structure invariants are not considered. However, PREFIX is known to give useful results on huge source programs.

Verification based on static analysis has culminated with shape analysis. The goals of the shape analyzer TVLA [148, 193, 147] are closest to ours but its approach is radically different. Rather than encoding programs in logic, TVLA performs fixed-point iterations on abstract descriptions of the store. Regarding precision and speed, PALE and TVLA seem to be at the same level. TVLA can handle some data abstractions and hence reason about sorting algorithms; we show in [161] that we can do the same. TVLA analyzes programs with only pre- and post-conditions, where PALE often uses loop invariants and assertions. This seems like an undisputed advantage for TVLA; however, not having invariants can cause a loss in precision making TVLA falsely reject a program. Regarding the specification of new data structures we claim an advantage. Once a graph type has been abstractly described with PAL, the PALE tool is ready to analyze programs. In TVLA it is necessary to specify in three-valued logic an operational semantics for a collection of primitive actions specific to the data structure in question. Furthermore, to guarantee soundness of the analysis, this semantics should be proven correct by hand. TVLA is applicable also to data structures that are not graph types, but so far all their examples have been in that class. Unlike PALE, TVLA cannot produce explicit counterexamples when programs fail to verify.

There exists a variety of assertion languages designed to express properties of data structures, such as ADDS [104], L_r [21], and Shape Types [91]. We rely on PAL since it provides a high degree of expressiveness while still having a decision procedure that works in practice.

2.2.3 Pointer Assertion Logic and Graph Types

The notion of graph types was introduced in [138] for formalizing the shapes of heap structures that go beyond being classical tree-shaped recursive data structures. Graph types allow many common heap shapes, such as doubly-linked lists or threaded trees, to be expressed concisely. A graph type is a recursive data type augmented with *aux*-

iliary pointers. The recursive data type defines a tree-shaped backbone. The auxiliary pointers are specified by *routing expressions* and are required to be functionally determined by the backbone. Routing expressions are regular expressions over a language of primitive directives that navigate around the backbone.

The decidability results in [138] are shown by an encoding of routing expressions into M2L(Tree). To express data structures in our programming language, we use an extension of graph types where the full M2L(Tree) logic is available as routing expression language. The regular expressions from [138] are included as syntactic sugar.

Our programming language is a simple imperative language with procedures. Its values consist of pointers and booleans. There are two kinds of pointers: regular pointers and *data pointers*. The former may point freely into the heap, while the latter at certain program places, called *cut-points*, must point to unique and disjoint graphs that match the declared graph types and span the entire heap. The latter property can be used to express leaking memory.

As an example, the following graph type declaration written in PALE syntax defines linked lists with tail pointers:

```
type Head = {
  data first: Node;
  pointer last: Node[this.first<next*. [pos.next=null]>last]
}
type Node = {
  data next: Node;
}
```

We abstract away the actual data contained in the lists and only consider the graph structure. The `last` field defines auxiliary pointers with an associated routing expression. This expression states that there is a path from the current record to the one pointed to by `last`. This path starts by the `first` pointer and then follows an arbitrary number of `next` pointers until a null pointer is encountered.

PALE programs can be annotated with formulas written in Pointer Assertion Logic which is a monadic second-order logic over graph types. In addition to the usual boolean connectives, it allows quantification over heap records, both of individual elements and of sets of elements. Also, it contains basic relations for reasoning about the pointer and boolean variables and fields that occur in the program.

Annotations occur as pre- and post-conditions for procedures, as loop invariants and procedure call invariants, and in special *assert* and *split* statements, expressing desired properties of the program store. The task of the verification procedure is to check that these properties are satisfied, that the graph types are maintained throughout execution, and that null pointer dereferences and memory leaks cannot occur.

The following example program implements the reverse operation for linked lists and is annotated with some correctness requirements:

```
type List = {data next:List;}

pred roots(pointer x,y:List, set R:List) =
  allpos p of List: p in R <=> x<next*>p | y<next*>p;
```



```

proc reverse(data list:List):List
  set R:List;
  [roots(list,null,R)]
  {
    data res:List;
    pointer temp:List;
    res = null;
    while [roots(list,res,R)] (list!=null) {
      temp = list.next;
      list.next = res;
      res = list;
      list = temp;
    }
    return res;
  }
  [roots(return,null,R)]

```

In the first line, the graph type `List` is defined. This particularly simple example does not use any auxiliary pointers. Then, a predicate `roots` is defined. Given two list pointers and a set of heap records, it evaluates to *true* if all records in the set are reachable from one of the pointers through a sequence of `next` fields. The `reverse` procedure essentially consists of a `while` loop that iteratively reverses the given list named `list`. Annotations are written in square brackets. The precondition of the procedure binds the logical variable `R` to be the set of reachable nodes from `list`, and the postcondition says that those nodes are reachable from the returned pointer when the procedure exits.

We cannot express that the resulting list is precisely the reverse of the original list, but since we have expressed the requirements that there are no null pointer dereferences or memory leaks, that no records are lost by the procedure and that the resulting structure is a proper list without loops, we have a fine-grained mask for errors.

2.2.4 Encoding Programs and Properties in MONA Logic

As mentioned, the transduction technique for encoding programs into MONA logic only works for loop-free code. Therefore, we resort to Hoare logic [106] for splitting the program into loop-free fragments connected by pre- and post-conditions and invariants. The resulting Hoare triples have a non-standard form: a PALE triple consists of declarations of logical variables, a precondition PAL formula, and a statement sequence. The statements are either assignments, conditionals, or assertions containing PAL formulas. In contrast to standard Hoare triples, these ones also allow assertions within the code and not only as postconditions.

In the PALE tool, the first phase splits the input program into such *transduction instructions*, a form of intermediate language corresponding to the Hoare triples. Continuing the `reverse` example from above, the body of the `while` loop is transformed into the following transduction instructions:

```

transduce
  set R:List;
  [roots(list, res, R) & list!=null]
  temp = list.next;

```

```

list.next = res;
res = list;
list = temp;
assert [roots(list, res, R)]

```

This directly corresponds to the premise of the proof rule for `while` loops in Hoare logic:

$$\frac{\{\phi \wedge B\} S \{\phi\}}{\{\phi\} \text{ while } (B) \{S\} \{\phi \wedge \neg B\}}$$

Once the program has been split into transduce constructs, each of them is verified independently by a linear encoding into MONA tree logic. Even for large programs, the loop-free fragments are typically relatively small. In this sense, our technique is highly modular, even though the resulting formulas may require heavy computations to verify. This is in contrast to program analysis approaches, which may require iteration over large program fragments.

The basic idea of the transduction technique is to represent the tree structured backbones of the heap structures by the tree models of M2L(Tree). Each data pointer in the program is allocated one tree structure. It is straightforward to represent multiple trees of non-binary fanout by a single WS2S or M2L(Tree) tree. As in [138] the auxiliary pointers of the graph types can be reduced to M2L(Tree) formulas on the backbone structures. We use the recursive-type parts of the declared graph types to automatically derive a GTA guide, which is crucial for the performance as explained in Section 2.1.5. The program variables are modeled as free variables, which are universally quantified in the final validity formula that is given to MONA. The transduction works by defining a family of *store predicates* for each program point:

- `bool_T_b(v)` gives the value of the `bool` field `b` in a record `v` of type `T`;
- `succ_T_d(v,w)` holds if the record `w` is reachable from the record `v` of type `T` along a data field named `d`;
- `null_T_d(v)` holds if the data field `d` in the record `v` of type `T` is null;
- `succ_T_p(v,w)` holds if the record `w` is reachable from the record `v` of type `T` along a pointer field named `p`;
- `null_T_p(v)` holds if the pointer field `p` in the record `v` of type `T` is null;
- `ptr_d(v)` holds if the record `v` is the value of the data variable `d`;
- `null_d()` holds if the data variable `d` is null;
- `ptr_p(v)` holds if the record `v` is the destination of the pointer variable `p`;
- `null_p()` holds if the pointer variable `p` is null;
- `bool_b()` gives the value of the boolean variable `b`;
- `memfailed()` holds if a null-pointer dereference has occurred.

For the entry point of the program fragment, these predicates essentially coincide with the basic predicates of the M2L(Tree) logic. As an example, the `succ_T_d(v,w)` predicate is initially defined by the basic successor predicate associated to the `d` field in the `T` type.

The effect of each individual statement is modeled by updating the store predicates accordingly. Assume that the statement

```
temp = list.next;
```

from the example above is enclosed by program points i and j . The predicates are transformed as follows:

```
memfailedj() = memfailedi() | null_listj()
ptr_tempj(v) = ex2 w: ptr_listi(w) & succ_Node_nextj(w, v)
null_tempj() = ex2 w: ptr_listi(w) & null_Node_next(w)
```

while the other store predicates remain unchanged. These transformations encode the behavior that there has been a memory error at j if there was one at i or `list` was null; at j , a pointer v refers to the same record as `temp` if the record referred to by `list` has a `next` field pointing to the v record; and similarly, `temp` is null at j if that `next` field was null.

Once the entire code fragment has been encoded as a formula, the correctness properties are verified by model checking

$$M \models \varphi$$

where M is the encoding of the program statements in the fragment and φ is an encoding of the precondition and the assertion formulas combined with the implicit requirements that there are no memory errors and the graph types are maintained, all expressed in MONA logic.

In [161], we make a number of experiments to investigate whether the technique works in practice. Our largest example is the insert procedure for red-black search trees, including the auxiliary procedures for performing rotations and non-balanced insertion. In total, this example consists of 150 lines packed with pointer operations. Reasoning about correctness of such a program is traditionally done by a manual or semi-automated proof. With PALE, we can encode the red-black tree structure as a simple graph type and its data type invariant by a Pointer Assertion Logic formula. Writing the loop invariants naturally requires knowledge about red-black search trees and can be hard to get right. However, we believe that the programs that require the most complicated invariants are also those that have the most complicated pointer operations and hence are the ones in most need of verification. The red-black invariant consists of three properties: 1) the tree root is black, 2) red nodes have only black children, and 3) for any two direct paths from the root to some leaf, the number of black nodes is the same. The first two can directly be expressed in PAL. The third is not expressible, but verifying that the other properties are satisfied, including the implicitly stated ones about null pointer dereferences, etc., we have a fine-grained filter for programming errors. PALE and MONA verify the correctness properties of the program in around one minute using 44 MB memory.

From this and the other experiments we conclude in [161] that in spite of the drawback that explicit invariants must be provided, quite detailed properties of pointer programs can be expressed in Pointer Assertion Logic and program verification with monadic second-order logic is feasible.

Chapter 3

Languages for Web Service Development

Since the invention of the Web around ten years ago, an increasing amount of Web services are becoming interactive. Generating Web pages dynamically in dialog with the client has the advantages of providing up-to-date and tailor-made information, and allowing two-way communication such that interactions can have side-effects on the server. The development of systems for constructing such dynamic Web services has emerged as a whole new research area.

The <bigwig>, **JWIG**, and DSD projects were initiated at BRICS (for DSD, in cooperation with AT&T Labs Research) to address some of the many different issues involved in Web service development. WWW home pages for these projects can be found at <http://www.brics.dk/bigwig/>, <http://www.brics.dk/JWIG/>, and <http://www.brics.dk/DSD/>, respectively. There, all relevant tools, examples, manuals, and articles are available.

The <bigwig> programming language has been designed by analyzing its application domain and identifying fundamental aspects of Web services, inspired by problems and solutions in existing Web service development languages. The core of the design consists of a session-centered service model together with a flexible template-based mechanism for dynamic Web page construction. Using specialized program analyses, certain Web specific properties are verified at compile time, for instance that only valid HTML 4.01 is ever shown to the clients. In addition, the design provides high-level solutions to form field validation, caching of dynamic pages, and temporal-logic based concurrency control, and it proposes syntax macros for making highly domain-specific languages. From the experience with <bigwig>, our language design has culminated with **JWIG**—a Java-based variant inheriting and improving the most successful features of <bigwig>.

A different aspect of Web services is the use of XML and the many surrounding technologies for representing and manipulating information in general. XML is relevant to Web services for several reasons: Web pages are written using the XML notation (or the preceding SGML notation); increasingly, XML is also being used for representing data on the servers and for data exchange between Web agents; and furthermore, most XML-based languages and tools are designed to operate on the Web. A central technology in the XML world is schema languages. The XML notation is by

itself essentially just a syntax for labeled trees. Schemas allow classes of XML documents to be defined, much in the same way as BNF grammars are being used to define the syntax of programming languages. Since the introduction of XML 1.0 in 1998 there have been numerous proposals for schema languages. However, most of these have been criticized, either for having too little expressive power or for being too complicated to learn and use. To accommodate this, we have developed yet another XML schema language, Document Structure Description (DSD), and a successor, DSD2. The latter is used in the **JWIG** program analysis to describe the syntax of XHTML 1.0, the XML variant of HTML 4.01. In fact, this analysis works for any XML language that can be described by DSD2.

In this chapter, we present the highlights of the contributions of the <bigwig>, **JWIG**, and DSD projects to the area of Web service development, and relate our solutions to existing technologies.

3.1 Interactive Web Services

The HTTP protocol, which is the foundation of the Web, is based on a client-server architecture where the client uses a browser to request a Web page or some other resource from the server. An interactive Web service is characterized by involving multiple interactions with each client, mediated by HTML forms in Web pages, and controlled by the service program running on the server.

Existing Web service programming languages in various ways provide only low-level solutions to problems specific to the domain of Web services. The overall ambitions of the <bigwig> and **JWIG** projects have been to identify the key areas of the Web service domain, analyze the problems with the existing approaches, and provide high-level solutions that will support development of complex services.

CGI [101] was the first platform for development of interactive Web services, based on the simple idea of letting a script generate the reply to incoming HTTP requests dynamically on the server, rather than returning a static HTML page from a file. Typically, the script is written in the general-purpose scripting language Perl, but any language supported by the server can be used. Being based on general-purpose programming languages, there is no special support for Web specific tasks, such as generation of HTML pages, and knowledge of the low-level details of the HTTP protocol are required. Also, HTTP/CGI is a stateless protocol that by itself provides no help in tracking and guiding users through series of individual interactions. This can to some degree be alleviated by libraries. In any case, there are no compile-time guarantees of correct runtime behavior when it comes to Web-specific properties, for instance ensuring that invalid HTML is never sent to the clients.

Servlets [201] is a popular higher-level Java-specific approach. Servlets, which are special Java programs, offer the common Java advantages of network support, strong security guarantees, and concurrency control. However, some significant problems still exist. Services programmed with servlets consist of collections of request handlers for individual interactions. Sessions consisting of several interactions with the same client must be carefully encoded with cookies, URL rewriting, or hidden input fields, which is tedious and error-prone even with library support, and it becomes hard to maintain an overview of large services with complex interaction flows. A second, although less

significant problem is that state shared between multiple client sessions, even for simple services, must be explicitly stored in a name–value map called the “servlet context” instead of using Java’s standard variable declaration scoping mechanism. Third, the dynamic construction of Web pages is not improved compared to CGI. Web pages are built by printing string fragments to an output stream. There is no guarantee that the result will always become valid HTML. This situation is slightly improved by using HTML constructor libraries, but they preclude the possibility of dividing the work of the programmers and the HTML designers. Furthermore, since client sessions are split into individual interactions that are only combined implicitly, for instance by storing session IDs in cookies, it is not possible to statically analyze that a given page sent to a client always contains exactly the input fields that the next servlet in the session expects.

JSP [202], ASP [107], PHP [9], and the countless homegrown variants were designed from a different starting point. Instead of aiming for complex services where all parts of the pages are dynamically generated, they fit into the niche where pages have mostly static contents and only small fragments are dynamically generated. A service written in one of these languages typically consists of a collection of “server pages” which are HTML pages with program code embedded in special tags. When such a page is requested by the client, the code is evaluated and replaced by the resulting string. This gives better control over the HTML construction, but it only gives an advantage for simple services where most of every page is static. JSP and Servlets are often used conjointly with Servlets taking care of the service logic and JSP pages producing the reply HTML pages.

The MAWL research language [8, 7, 144] was designed especially for the domain of interactive Web services. One innovation of MAWL is to make client sessions explicit in the program logic. Another is the idea of building HTML pages from templates. A MAWL service contains a number of sessions, shared data, and HTML templates. Sessions serve as entry points of client-initiated session threads. Rather than producing a single HTML page and then terminating as CGI scripts or Servlets, each session thread may involve multiple client interactions while maintaining data that is local to that thread. An HTML template in MAWL is an HTML document containing named gaps where either text strings or special lists may be inserted. Each client interaction is performed by inserting appropriate data into the gaps in an HTML template and then sending it to the client, who fills in form fields and submits the reply back to the server.

The notions of sessions and document templates are inherent in the MAWL language, and, being compilation-based, it allows important properties to be verified statically without actually running the service. Since HTML documents are always constructed from the templates, HTML validity can be verified statically. Also, since it is clear from the service code where execution resumes when a client submits form input, it can be statically checked that the input fields match what the program expects. One practical limitation of the MAWL approach is that the HTML template mechanism is quite restrictive, as markup cannot be inserted into the template gaps.

By studying services written in any of these preexisting languages, some other common problems show up. First of all, often surprisingly large portions of the service code tend to deal with form input validation. Client-server interaction takes place mainly through input forms, and usually some fields must be filled with a certain kind

of data, perhaps depending on what has been entered in other fields. If invalid data is submitted, an appropriate error message must be returned so that the client can try again. All this can be handled either on the client-side typically with JavaScript [87], in the server code or with a combination. In any case, it is tedious to encode.

Second, one drawback of dynamically generated Web pages compared to static ones is that traditional caching techniques do not work well. Browser caches and proxy servers can cause major improvements in saving network bandwidth, load time, and clock cycles, but when moving towards interactive Web services, these benefits disappear.

Third, most Web services act as interfaces to underlying databases that for instance contain information about customers, products, and orders. Accessing databases from general-purpose programming languages where database queries are not integrated requires the queries to be built as text strings that are sent to a database engine. This means that there is no static type checking of the queries. As known from modern programming languages, type systems allow many programming bugs to be caught at compile time rather than at runtime, and thereby improve reliability and reduce development cost.

Fourth, since running Web services contain many concurrently executing threads and they access shared information, for instance in databases on the server, there is a fundamental need for concurrency control. Threads may require exclusive access to critical regions, be blocked until certain events occur, or be required to satisfy more high-level behavioral constraints. All this while the service should run smoothly without deadlocks and other abrupt obstacles. Existing solutions typically provide no or only little support for this, for instance via low-level semaphores as Perl or synchronized methods in Servlets. This can make it difficult to guarantee correct concurrent execution of entire services.

Finally, since Web services usually operate on the Internet rather than on secure local networks, it is important to protect sensitive information, both from hostile attacks and from programming leaks. A big step forward is the Secure Sockets Layer (SSL) protocol [93] combined with HTTP Authentication [22]. These techniques can ensure communication authenticity and confidentiality, but using them properly requires insight into technical protocol and implementation details. Furthermore, they do not protect against programming bugs that unintentionally leak secret information. The “taint mode” in Perl offers some solution to this. However, it is runtime based, so no compile-time guarantees are given. Also, it only checks for certain predefined properties, and more specialized properties cannot be added.

Motivated by the languages and problems described above we have identified the following areas as key aspects of Web service development:

- *sessions*: the underlying paradigm of interactive Web services;
- *dynamic documents*: HTML pages must be constructed in a flexible, efficient, and safe fashion;
- *concurrency control*: Web services consist of collections of processes running concurrently and sharing resources;
- *form field validation*: validating user input requires too much attention from Web programmers so a higher-level solution is desirable;

- *database integration*: the core of a Web service is often a database with a number of sessions providing Web access; and
- *security*: to ensure authenticity and confidentiality, regarding both malicious clients and programming bugs.

To attack the problems, we have designed from scratch the `<bigwig>` language as a descendant of the MAWL language, and later the **JWIG** language as descendant of `<bigwig>`. Both `<bigwig>` and **JWIG** are high-level domain-specific languages [214], meaning that they employ special syntax and constructs that are tailored to fit their particular application domain and allow specialized program analyses, in contrast to library-based solutions.

The following sections give an overview of our results. In [40], we present the motivation and the design of `<bigwig>` in further detail; the paper [38] focuses on the session-based runtime system; and [37] explains our PowerForms language for making declarative form input validation. In another paper, [195], the DynDoc language for dynamic construction of Web pages is presented. We describe in [36] a caching technique for such dynamically constructed pages, and in [39] we show a program analysis which can statically check that only valid HTML 4.01 or XHTML 1.0 pages are constructed. In [57], we explain how the features of `<bigwig>` are integrated into Java, resulting in the **JWIG** language. In [58] we go into more detail of the notion of summary graphs used in the **JWIG** program analyses, relates it to the regular expression types from the XDuce language, and proposes some extensions to **JWIG** for deconstructing XML values. In [132, 133] we introduce the Document Structure Description language for making schemas for XML documents. A more recent variant of this language, DSD2, is in [57] used in the static analysis of **JWIG** programs. For a general introduction to Web service programming and to XML and the related technologies, see our online tutorials [163, 162].

3.2 The Session-Centered Approach

There is a variety of techniques for implementing interactive Web services, but they can be divided into three main paradigms: the *script-centered*, the *page-centered*, and the *session-centered*. Each is supported by various tools and suggests a particular set of concepts inherent in Web services.

3.2.1 Script-Centered Languages

The script-centered approach builds directly on top of the plain, stateless HTTP/CGI protocol. A Web service is defined by a collection of loosely related scripts. A script is executed upon request from a client, receiving form data as input and producing HTML as output before terminating. Individual requests are tied together by explicitly inserting appropriate links to other scripts in the reply pages.

A prototypical scripting language is Perl, but almost any programming language has been suggested for this role. CGI scripting is often supported by a large collection of library functions for decoding form data, validating input, accessing databases, and realizing semaphores. Even though such libraries are targeted at the domain of Web

services, the language itself is not. A major problem is that the overall behavior is distributed over numerous individual scripts and depends on the implicit manner in which they pass control to each other. This design complicates maintenance and precludes any sort of automated global analysis, leaving all errors to be detected in the running service [86, 7].

HTML documents are created on the fly by the scripts, typically using print-like statements. This again means that no static guarantees can be issued about their correctness. Furthermore, the control and presentation of a service are mixed together in the script code, and it is difficult to factor out the work of programmers and HTML designers [68].

The Java Servlets language also fits this category. The overall structure of a service written with servlets is the same as for Perl. Every possible interaction is essentially defined by a separate script, and one must use cookies, hidden input fields, or URL rewriting techniques to connect sequences of interactions with the clients—however, Servlets provide a session tracking API that hides many of the details. Many servlet servers use cookies if the browser supports them, but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled.

3.2.2 Page-Centered Languages

The page-centered approach is covered by languages such as ASP, PHP, and JSP, where the dynamic code is embedded in the HTML pages. In a sense, this is the inverse of the script-centered languages where HTML fragments are embedded in the program code. When a client requests a page, a specialized Web server interprets the embedded code, which typically produces additional HTML snippets while accessing a shared database. This approach is often beautifully motivated by simple examples, where pages are mainly static and only sporadically contain computed contents. For example, a page that displays the time of day or the number of accesses clearly fits this mold.

As long as the code parts only generate strings without markup it is easy to statically guarantee that all pages shown are valid HTML and other relevant properties. But as the services become more complex, the page-centered approach tends to converge towards the script-centered one. Instead of a mainly static HTML page with some code inserted, the typical picture is a single large code tag that dynamically computes the entire contents.

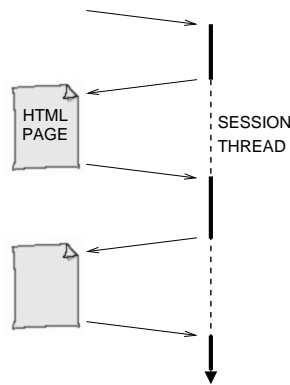
The JSP language is based on Java and is closely related to Servlets. Implementations work by compiling each JSP page into a servlet using a simple transformation. The ASP and PHP languages are very reminiscent of JSP. ASP is tied to Microsoft's Internet Information Server, although other implementations exist. Instead of being based on Java, it defines a language-independent connection between HTML pages and scripting languages, typically either VisualBasic Script or Microsoft's version of JavaScript. PHP is a popular Open Source variant whose scripting language is a mixture of C, Java, and Perl.

These languages generally provide only low-level support for tracking client sessions and maintaining session state. Cookies, hidden input fields, and some library support is the common solution. Also for other Web service aspects, such as databases and security, there is often a wide range of libraries available but no direct language support.

3.2.3 Session-Centered Languages

The pure session-centered approach was pioneered by the MAWL project. Here a service is viewed as a collection of distinct *sessions* that access some shared data. A client may initiate a session *thread*, which is conceptually a process running on the server. Interactions with the client are viewed as remote procedure calls from the server, as known from classical construction of distributed systems but with the roles reversed.

The flow of an entire session is programmed as a single sequential program, which is closer to ordinary programming practice and offers the compiler a chance to obtain a global view of the service. This flow can be illustrated as follows:



On the left is the client's browser, and on the right is a session thread running on the server. The thread is initiated by a client request and controls the sequence of interactions.

Important issues such as concurrency control and the connection between HTML forms being shown and field values being received become simpler to understand in this context and standard programming solutions, such as data-flow analysis, are more likely to be applicable.

The overall structure of `<bigwig>` and **JWIG** programs is directly inspired by MAWL. A program in these languages contains a complete specification of a Web service. Such a service contains a collection of named *sessions*, each of which essentially is an ordinary sequential program. A client has the initiative to invoke a thread of a given session, which is a process on the server that executes the corresponding sequential code and exclusively communicates with the originating client. Communication is performed by *showing* the client an HTML page, which implicitly is made into a form with an appropriate URL return address. While the client views the given document, the session thread is suspended on the server. Eventually the client submits the form, which causes the session thread to be resumed and any form data entered by the client to be *received* into program variables. A simple `<bigwig>` service is the following:

```
service {
  html hello = <html>Enter your name: <input name=handle></html>;
  html greeting =
    <html>Hello <[who]>, you are user number <[count]></html>;
  html goodbye = <html>Goodbye <[who]></html>;

  shared int users = 0;
```

```

session Hello() {
    string name;
    show hello receive[name=handle];
    users++;
    show greeting<[who=name,count=users];
    show goodbye<[who=name];
}
}

```

This service defines the code for Hello sessions. Such a session starts by showing the hello page to the client, who fills in and submits his name. That value is on the server received into the variable *name*. Then the number of users is incremented, a greeting is produced, and finally, the goodbye page is shown. Variables that are declared with the modifier *shared* are shared between all the running session threads. For other variables, each thread has a local instance. The access to the shared *users* variable actually needs concurrency control to avoid race conditions; this topic is described in Section 3.6.1.

The program structure is basically as in MAWL. However, <bigwig> provides a number of new features. Most importantly, HTML templates are now *first-class values*. That is, HTML is a built-in data type, written *html*, and its values can be passed around and stored in variables as any other data type. Also, the HTML templates are *higher-order*, meaning that instead of only allowing text strings to be inserted into the template gaps, we also allow insertion of other templates. This is done with the special *plug* operator, *x*<[*g*=*y*], which inserts a string or template *y* into the *g* gaps of the *x* template, as explained in Section 3.3.

For comparison, we show a similar program written in the **JWIG** language:

```

import dk.brics.jwig.runtime.*;

public class MyService extends Service {
    int users = 0;
    synchronized int next() { return ++users; }

    public class ExampleSession extends Session {
        XML wrapper =
            [[ <html><head><title>JWIG</title></head>
              <body><[body]></body></html> ]];
        XML hello =
            [[ <form>Enter your name: <input name="handle">
              <input type="submit"></form> ]];
        XML greeting =
            [[ Hello <[who]>, you are user number <[count]> ]];
        XML goodbye =
            [[ Goodbye <[who]> ]];

        public void main() throws IOException {
            show wrapper<[body=hello];
            String name = receive handle;
            show wrapper<[body=greeting<[who=name,count=next()]];
            exit wrapper<[body=goodbye<[who=name]];
        }
    }
}

```

JWIG is a variant of Java with special constructs for producing and showing Web pages and receiving form input. A service is defined as a subclass of `Service` from the package `dk.brics.jwig.runtime` and a session is an inner class being a subclass of `Session`. Fields that are declared in the outer service class are shared between all session threads of the service. **JWIG** uses the XML variant of HTML, XHTML, but construction of documents is essentially the same as in `<bigwig>` as there are only minor notational differences between HTML and XHTML. The Java language is a natural choice for developing modern Web services. As previously mentioned, its built-in network support, strong security guarantees, concurrency control, and widespread deployment in both browsers and servers, together with popular development tools make it relatively easy to create Web services. An advantage of **JWIG** compared to `<bigwig>` is the vast amount of Java packages that are immediately available. Specifically, they make it easier to write the many parts of typical Web services that manipulate data without having anything to do with the Web in particular, such as using data containers, text operations, etc. The lack of such packages for `<bigwig>` was increasingly becoming an impediment when writing larger applications. Another advantage is that the program analyses in **JWIG** have been improved notably in various ways compared to `<bigwig>`, as described in [57].

The session-centered approach that we use in `<bigwig>` and **JWIG** has two essential benefits compared to other approaches: First of all, we believe that the control-flow of a service written in a language with inherent session support becomes clearer to the programmer and hence simplifies development and maintenance of the Web services. Second, the compiler also has a better chance of understanding this flow. Specifically, `<bigwig>` and **JWIG** services are verified at compile time to ensure that 1) a plug operation always finds a gap with the specified name in the given fragment, 2) the code that receives form input is presented with the expected fields, and 3) only valid HTML 4.01 or XHTML 1.0 is ever sent to the clients.

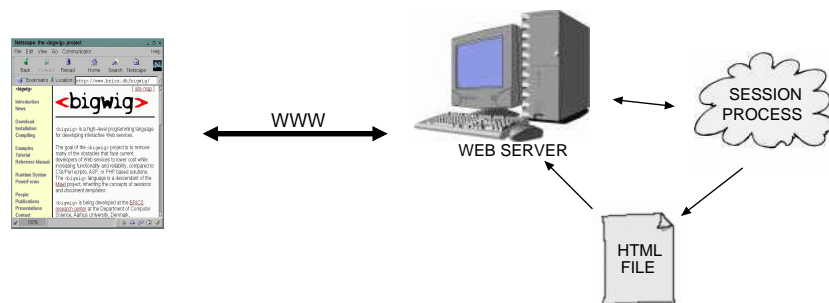
3.2.4 A Runtime Model with Support for Sessions

The session-based model can be implemented on top of the CGI protocol. One naive approach is to create session threads as CGI scripts where all local state is stored on disk. At every session interaction, the thread must be started again and restore its local state, including the call stack, in order to continue execution. A better approach is to implement each session thread as a process that runs for the whole duration of the session. For every interaction, a tiny transient CGI script called a *connector process* is executed, acting as a pipe between the Web server and the session process. This approach is described in detail in [38]. Our newest implementation is instead based on a specialized module for the Apache Web server. Naturally, this is much faster than the CGI solutions since it does not create a new process for every single interaction: In `<bigwig>`, each session thread is associated with one system process, and in **JWIG**, each service runs as one JVM with a Java thread for each session.

Two common sources of problems with standard implementations of sessions are history buffers and bookmarking features found in most browsers. With history buffers and the “back” button, the users can step back to a previous interaction, and either intentionally or unintentionally resubmit an old input form. Sometimes this can be a useful feature, but more often it causes confusion and annoyance to the users who

may, for instance, order something twice in online shopping systems. It is a general problem that the information shown to the user in this way can be obsolete, because it is tailor-made only for the exact time of the initial request. Since the information was generated from a shared database that may have changed entirely, it does generally not make sense to “step back in time” using the history buffer. It also becomes hazardous to try to use bookmarks to temporarily suspend a session. Invoking the bookmark will typically cause a CGI script to be executed a second time instead of just displaying its results again.

Our runtime system for **<bigwig>** and **JWIG** provides a simple but unique solution to these problems: Each session thread is associated with a URL which points to a file on the server containing the latest HTML page shown to the client. Instead of sending the contents directly to the client at every `show` statement, we redirect the browser to this URL:



Since the URL serves as the identification of the session thread, this solves the problems mentioned above: The history list of the browser now only contains a single entry for the duration of the session, the sessions can now be bookmarked for later use, and in addition, the session identity URL can be passed around manually—to another browser, for instance—without problems. When using URLs instead of cookies to represent the session identity, it also becomes possible for a single user to simultaneously run multiple sessions in different windows but with the same browser.

Furthermore, with this simple solution we can automatically provide the client with feedback while the server is processing a request. This is done by, after a few seconds, writing a temporary response to the HTML file, which informs the client about the status of the request. This temporary file reloads itself frequently, allowing for updated status reports. When the final response is ready, it simply overwrites the temporary reply file, causing the reloading to stop and the response to be shown. This simple technique may prevent the client from becoming impatient and abandoning the session.

3.3 Dynamic Construction of Web Pages

Constructing the Web pages that are shown to the users is obviously among the most central activities in Web services. With the script-centered approach, for instance using CGI or Servlets, the pages are usually constructed by printing HTML fragments to an output stream, as explained above. This is a highly flexible mechanism, except that the pages must be constructed in a linear fashion from top to bottom instead of being

composed in a more logical manner. However, these languages permit essentially no static guarantees, for instance about validity of the generated HTML pages.

In MAWL, all HTML templates are placed in separate files and viewed as procedures of a kind, with the arguments being strings that are plugged into gaps in the template and the results being the values of the form fields that the template contains. This allows a complete separation of the service code and the HTML code. Thereby certain static guarantees are possible and the work of programmers and HTML designers can be separated to increase their productivity. A disadvantage is that this template mechanism becomes too rigid compared to the flexibility of the script-centered languages.

The page-centered approach known from JSP and the related languages can conceptually be placed between the script-centered and the session-centered. If JSP pages are mostly static HTML, then they are reminiscent of the page templates in MAWL, except that there is no restriction on what may be inserted into the gaps. With such pages, it is, for instance, relatively easy to argue that only valid HTML is produced. However, in the other extreme, a JSP page consisting entirely of code is essentially the same as a Servlet program. A more detailed comparison of how Web pages are constructed in the various languages can be found in [40, 57].

In the following, we focus on dynamic construction of Web pages in the **JWIG** language. The language constructs in `<bigwig>` only vary slightly compared to **JWIG**, but the program analyses are structured differently.

Web pages are constructed from XML *templates*. A template is a well-formed XML fragment which may contain named *gaps*. A special *plug* operation is used to construct new templates by inserting existing templates or strings into gaps in other templates. Templates are identified by a special data type, XML, and may be stored in variables and passed around as any other type. Once a complete XHTML document has been built, it can be used in a `show` statement.

Syntactically, the **JWIG** language introduces the following new expressions into Java for dynamic XML document construction:

<code>[[xml]]</code>	(template constant)
<code>exp₁ <[g = exp₂]</code>	(the plug operator)
<code>([[xml]]) exp</code>	(XML cast)
<code>get url</code>	(runtime template inclusion)

These expressions are used to define template constants, plug templates together, cast values to the XML type, and to include template constants at runtime, respectively. The *url* denotes a URL of a template constant located in a separate file, and *xml* is a XML template according to the following grammar:

<code>xml</code>	<code>: str</code>	(character data)
	<code> <name atts> xml </name></code>	(element)
	<code> <[g]></code>	(template gap)
	<code> <{ stm }></code>	(code gap)
	<code> xml xml</code>	(sequence)
<code>atts</code>	<code>: ε</code>	(empty)
	<code> name="str"</code>	(attribute constant)

<i>name</i> =[<i>g</i>]	(attribute gap)
<i>atts atts</i>	(sequence)

Here *str* denotes an arbitrary Unicode string, *name* an arbitrary identifier, *g* a gap name, and *stm* a statement block that returns a value of type `String` or `XML`.

XML templates can be composed using the plug operation $exp_1 <[g = exp_2]$. The result of this expression is a copy of exp_1 with all occurrences of the gap named *g* replaced by copies of exp_2 .

Code gaps are special gaps that directly contain code. When a template containing code gaps is shown, the code blocks in the code gaps are executed in document order. The resulting strings or templates are then inserted in place of the code. With code gaps, it is easy to emulate the page-centered approach of constructing Web pages.

Communication with the client is performed through the `show` statement which takes as argument an XML template to be transmitted to the client. The responses from the client are subsequently obtained using the `receive` expression, which takes as argument the name of an input field in the XHTML document that was last shown to the client and returns the corresponding value provided by the client as a `String`. There may be several occurrences of a given input field. In this case, all the corresponding values may be received in order of occurrence in the document into a `String` array using the variant `receive[]`.

We define *DynDoc* as this sublanguage of **JWIG** that deals with document construction. *DynDoc* combines the best of each of the alternatives: Web pages can be composed in a logical order and all HTML tags are automatically balanced, in contrast to using a script-centered language; with code gaps, the page-centered approach of embedding code in Web pages can be emulated directly; the templates can be separated from the main code, as in MAWL; by allowing templates to be plugged into templates to construct new templates it is substantially more flexible than MAWL; and finally, *DynDoc* allows a number of compile-time guarantees that are otherwise only seen in the simpler MAWL language. The domain-specific program analyses that are used to provide these guarantees are the topic of the next section.

As an extra benefit, *DynDoc* also proposes a solution to another problem: caching of dynamically constructed Web pages. Traditional Web caching based on HTTP works by associating an expiration time to all files sent by the servers to the clients and caching the files within the browsers. This has helped in decreasing both network and server load and response times. However, the technique was designed primarily for files whose contents rarely or never changes, not for documents dynamically generated by interactive Web services. The gradual change from statically to dynamically generated documents has therefore caused the impact of Web caching to degrade. With *DynDoc* we have a unique opportunity to automatically separate the template constants from the dynamic parts of the generated Web pages and thereby store the templates individually and effectively revive the browser caches. In the context of <bigwig> we describe such a technique in [36].

The original version of the *DynDoc* language was introduced in [195]. More details about the newest versions used in <bigwig> and **JWIG** can be found in [40] and [57].

3.4 Program Analyses

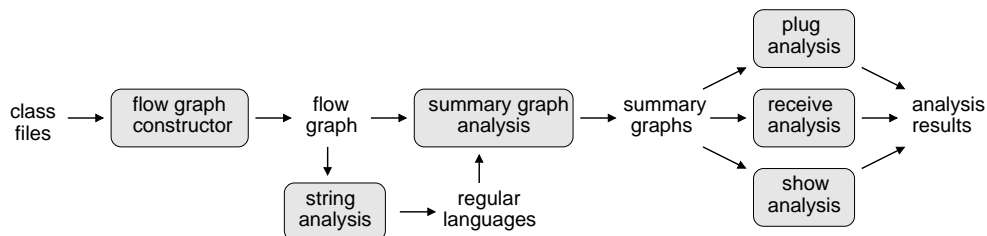
The high-level approach of dynamically constructing Web documents from well-formed fragments in `<bigwig>` and **JWIG** makes it possible to perform some highly specialized program analyses for the particular application domain of dynamic document construction. The overall goal of these analyses is to allow as liberal use of dynamic documents as possible while guaranteeing that no errors occur. More precisely, our analyses in **JWIG** provide the following guarantees for a given program:

- *plug consistency*: that gaps are always present when subjected to the plug operation and XML templates are never plugged into attribute gaps;
- *receive consistency*: that input fields occur the right number of times in the shown documents so `receive` and `receive[]` operations always succeed; and
- *show validity*: that all documents being shown are valid XHTML 1.0 [182].

If any of these correctness properties is not satisfied at runtime, an exception will be thrown. The analyses try to verify at compile time that these exceptions cannot occur, and if they can occur, an explanatory warning message is automatically produced.

As for all useful program analyses, ours are incomplete, meaning that correct programs may occasionally be falsely rejected. According to Rice's well-known theorem, the problem of obtaining precise answers to interesting questions about the behavior of programs written in a Turing complete formalism is always undecidable. Thus, approximations are inevitable. However, our analyses are designed to be conservative, meaning that they err only on the safe side: a program that can fail is never accepted as correct. The difficulty in designing such analyses is to make them sufficiently precise and at the same time sufficiently fast to be practically useful.

Our **JWIG** analyses work as follows. First, the special syntactic constructs of **JWIG** are translated into appropriate Java constructs by a simple source-to-source desugaring transformation, and then the resulting Java source files are compiled into Java class files by a standard Java compiler. The analysis works on the class file level. From the class files, we first generate *flow graphs*. From these, we generate *summary graphs*, which we analyze in three different ways corresponding to the properties mentioned above. In order to produce the summary graphs, we first need a preliminary *string analysis* of the flow graphs. The structure of the whole analysis on the class files is illustrated in the following figure:



Although the theoretical worst-case complexity of the analyses is $O(n^6)$ where n is the size of the given **JWIG** program, we are able to show by a number of benchmark tests

in [57] that the technique works well in practice. The precision of the analyses seems adequate since no false errors were encountered.

For programs where verification fails, useful warnings are provided, for example: “*field `howmany` may not be available for the receive operation on line 87*”, or “*invalid XHTML may be shown on line 117; the following schema requirements are not satisfied: [...]*”. Clearly, such detailed feedback aids the programmer in debugging the program.

The structure of the analysis of <bigwig> programs is slightly different from the one presented here for **JWIG**. Instead of basing all the analyses on summary graphs, we there perform a data-flow analysis on the flow-graph level to check for plug and receive consistency, as explained in [195]. In [57] we describe the benefits of using summary graphs for all the analyses.

3.4.1 Flow Graphs for JWIG Programs

Given a **JWIG** program, we first construct an abstract flow graph as a basis for the subsequent data-flow analyses. The flow graph captures the flow of string and XML template values through the program and their uses in show, plug, and receive operations, while abstracting away other aspects of the program behavior.

The nodes in a flow graph correspond to abstract statements, such as assignments, or show or receive operations. These statements contain expressions, for instance plug operations, constant XML templates, and variable reads. There are two kinds of edges: a *flow edge* models the data flow between the program points and is labeled with a set of program variables indicating which values are allowed to flow along that edge; a *receive edge* goes from a receive node to a show node, indicating that the control-flow of the program may lead from the corresponding show statement to the receive statement without reaching another show first. In [57] we formally define the semantics of flow graphs as solutions to constraint systems.

Constructing the flow graph of a given **JWIG** program is quite technical due to the many features of the Java language that need to be considered. The approach described in [57] proceeds in eight phases. First, each method body is translated individually. Then code gaps, method invocations, exceptions, show and receive operations, arrays, and field variables are handled specially. Finally, some graph simplifications are made where all flow edges that have been constructed in the previous phases are replaced by definition–use edges, each labeled with only a single variable. The resulting flow graph contains all the information we need from the original program to provide the desired guarantees of the program behavior.

3.4.2 Summary Graphs for XML Expressions

We now perform a *summary graph analysis* on the flow graph generated for the given **JWIG** program. Summary graphs model how templates are constructed and used at runtime. This analysis depends on a preliminary *string analysis* that for each string expression finds a regular language that approximates the set of strings it may evaluate to. For each of these two analyses we define a lattice expressing an abstraction of the data values along with a corresponding abstract semantics of the expressions and statements, and then apply standard data-flow analysis techniques to find the least so-

lutions. The result is that each XML expression occurring in the program is associated with a summary graph describing its possible values.

The string analysis which is currently applied is rather simple. It merely tracks the propagation of string constants throughout the program, using the set of all possible strings to model the results of `receive` and other string operations.

The lattice and transfer functions for the summary graph analysis are more involved. Let G be the set of gap names that occur in the program and N be a set of *template indices* denoting the instances of XML template constants. A *summary graph* SG is a finite representation of a set of XML documents and is defined as follows:

$$SG = (R, T, S, P)$$

where

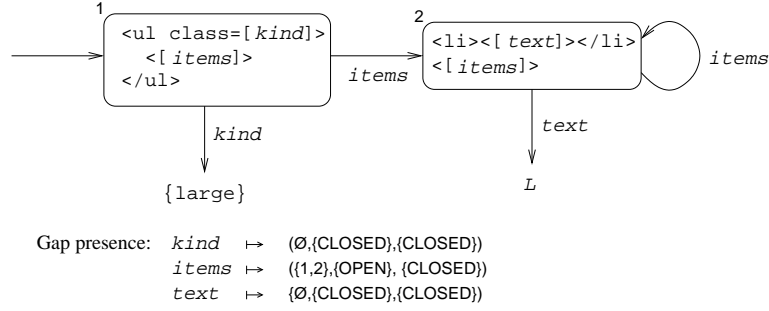
$R \subseteq N$ is a set of *root nodes*,
 $T \subseteq N \times G \times N$ is a set of *template edges*,
 $S : N \times G \rightarrow REG$ is a *string edge* map, and
 $P : G \rightarrow 2^N \times \Gamma \times \Gamma$ is a *gap presence* map

using the following definitions:

REG is the set of regular languages over the Unicode alphabet, and
 $\Gamma = 2^{\{OPEN, CLOSED\}}$ is the *gap presence lattice* whose ordering is set inclusion.

This notion of summary graphs is the cornerstone of our program analyses. The nodes correspond to the constant XML templates that appear in the program, and the edges result from the plug operations. Intuitively, the language of a summary graph is the set of XML documents that can be obtained by unfolding its templates, starting from a root node and plugging templates and strings into gaps according to the edges. The presence of a template edge $(n_1, g, n_2) \in T$ informally means that the n_2 template may be plugged into the g gaps in the n_1 template, and a string edge $S(n, g) = L$ means that every string in the regular language L may be plugged into the g gaps in the n template. The gap presence map, P , specifies for each gap named g which template constants may contain open g gaps reachable from a root and whether g gaps may or must appear somewhere in the unfolding of the graph, either as template gaps or as attribute gaps. The first component of $P(g)$ denotes the set of template constants with open g gaps, and the second and third components describe the presence of template gaps and attribute gaps, respectively. The value OPEN means that the gaps may be open, and CLOSED means that they may be closed or never have occurred. We need this gap presence information to 1) determine where edges should be added when modeling plug operations, 2) model the removal of gaps that remain open when a document is shown, and 3) detect that plug operations may fail because the specified gaps are not present.

This unfolding of summary graphs into sets of concrete XML documents is formally defined in [57] where the transfer functions for the analysis are also shown. The following illustration shows an example summary graph:



The language of this summary graph is the set of XML documents that consist of `ul` elements with a `class="large"` attribute and zero or more `li` items containing some text from the language L . In general, the summary graphs that are constructed are conservative, since they may denote languages that are too large. This means that the subsequent analyses can be sound but not complete.

A variant of the summary graphs shown here was introduced in [39] for verifying HTML validity of documents in `<bigwig>`. We show in the paper [58] that the notion of summary graphs is expressively equivalent to the regular expression types from XDuce, if disregarding attributes and string edges. That paper also suggests how the DynDoc language can be extended with operations for deconstructing XML values while still being able to provide static guarantees based on summary graphs.

The remaining analyses described in the following sections are independent of both the original **JWIG** program and the flow graphs. All the relevant information is at this stage contained in the inferred summary graphs.

3.4.3 Analyzing Plug Operations

For each plug operation, $x \leftarrow [g=y]$, occurring at some program point ℓ in the given **JWIG** program, we have constructed a summary graph. From this graph it is possible to verify whether g gaps are always present in the value of x at ℓ , and, in case y is an XML expression and not a string expression, that none of the present g gaps are attribute gaps. The desired information is directly available in the gap presence map component of the summary graph, so a trivial inspection of this map suffices to check plug consistency.

3.4.4 Analyzing Receive Operations

We now validate the `receive` and `receive[]` operations. For the single-string variant, `receive`, it must be the case that for all program executions, the last document being shown before the `receive` operation contained exactly one field of the given name. Also, there must have been at least one `show` operation between the initiation of the session thread and the `receive` operation. The array variant, `receive[]`, always succeeds, so technically, we do not have to analyze those operations. However, we choose to consider it as an error if we are able to detect that for a given `receive[]` operation, there are no possibility of ever receiving other than the empty array.

Given a specific `receive` operation, we need to count the number of occurrences of input fields of the given name that may appear in every document sent to the client in an associated `show` operation. We use a lattice structure with the following elements:

“0” means that there are always zero occurrences of the field, “1” means that there is always exactly one occurrence, “*” means that the number varies depending on the summary graph unfolding or that it is greater than one, “◇” represents one or more radio buttons, and “⊥” represents an unknown number. We need to count radio buttons specially, because only one value is submitted for a whole list of such buttons having the same name. A conservative approximation of the desired information can be extracted from the receive edges in the flow graph and the summary graphs of the associated show operations. For each of these summary graphs, we define a constraint system and solve it using fixed-point iteration. The details can be found in [57].

3.4.5 Analyzing Show Operations

For every show statement in the **JWIG** program, we have computed a summary graph that describes how the XML templates are combined in the program and which XML documents may be shown to the client at that point. This gives us an opportunity to verify that all documents being shown are *valid* relative to a desired document type, meaning that certain syntactic requirements are satisfied. In particular, we ensure that the documents are valid XHTML 1.0. Currently, no other Web service programming language than `<bigwig>` and **JWIG** allows such compile-time guarantees to be given while at the same time providing a flexible mechanism for dynamically constructing documents.

As described in Section 3.7, there exist many formalisms, called schema languages, for describing the syntax of XML-based languages. Traditionally, DTD has been used, but its limited expressiveness has motivated the development of more advanced formalisms. For the analysis of show operations in **JWIG**, we use DSD2, a successor to the Document Structure Description (DSD) language. For instance, the DSD2 schema for XHTML 1.0 describes several syntactic requirements that cannot be expressed in the DTD language.

The semantics of most schema languages are defined in terms of a top-down traversal of a given XML document, where satisfaction of the syntactic requirements specified in the schema is verified for each element in turn. Recall that a summary graph is a symbolic representation of a set of XML documents. The basic idea for validating all the documents represented by a summary graph is to generalize the top-down algorithm for validating individual documents, intuitively by unfolding the graph while performing memoization to ensure fast termination. Section 3.7 gives an overview of the DSD and DSD2 languages and of the algorithm for validating a summary graph with respect to a DSD2 schema.

3.5 Declarative Form Field Validation

With traditional Web service languages, a considerable amount of code is expended on checking whether the data supplied by users in the form input fields is of the right form, and when it is not, producing error messages and requesting the fields to be filled in again. For example, some fields must contain only digits, and others are required to be valid email addresses or dates. Often, validation requirements for one field depend on what the user has selected in other fields. The main problem is to program solutions

that are robust to modifications in the service logic and in the client environment, efficient on both the client and the server, and at the same time user friendly.

Using the *server-side* approach, where the validation is done entirely on the server, raises two kinds of problems: 1) every program part that sends a page to a client must be wrapped into a loop that repeats until the input is valid, so the main service code tends to be cluttered with code handling input validation; and 2) the types of user interactions in case of invalid input are limited, since errors are not detected until the client attempts to submit, and similarly, menu options cannot be made to appear or disappear dynamically depending on the input.

The alternative approach is to perform the validation on the *client-side*, typically using JavaScript [87]. This separates the validation code from the main service code and also has the benefit that validation can be performed incrementally as the user enters the data using the many graphical features available through JavaScript. There still are problems though: Finding the subset of JavaScript that works properly on all browsers can be quite challenging. Often, this makes programmers give up or stop at very simple solutions. Furthermore, a more fundamental problem is that JavaScript is a general-purpose and imperative language, exposing the programmer to many unnecessary details and choices. Even though most of the JavaScript code that exists is code dealing with input validation, this is not precisely what the language was designed for.

An additional source of complexity is that client-side validation should not be used alone. The reason is that the client is perfectly capable of bypassing the JavaScript code, so an additional server side validation must always be performed. Thus, the same code must essentially be written both in JavaScript and in the server scripting language. All in all, programming form field validation is not as simple as it may appear, and traditional Web service programming languages provide no particular support, except perhaps for regular expression matching.

3.5.1 The PowerForms Language

Our solution is to include a small high-level domain-specific declarative sublanguage, called *PowerForms*, tailored to specification of form input validation requirements. From this language, a compiler automatically generates the JavaScript code, in the right subset that most browsers agree on, together with server-side code for performing the double-check.

In the following we briefly describe the PowerForms language. More details and examples can be found in [37, 190]. The implementation, made by M. Ricky, is available both as a stand-alone C package and as an integral part of the <bigwig> system. Lately, we have integrated a Java version of PowerForms with the **JWIG** system. This version is based on the `dk.brics.automaton` Java package for automata and regular expression operations [171].

Being a domain-specific language, PowerForms only provides constructs that directly correspond to central concepts in the problem domain, namely the domain of field input validation. The PowerForms language is based on the idea of associating regular expressions to the input fields for defining the valid values. We have chosen regular expressions since this formalism in numerous other applications has proven suitable for concise and declarative specification of such sets of strings. Typical formats, such as email addresses, URLs, dates, and ISBN numbers are all regular

languages.

The regular expressions are translated into minimal deterministic finite-state automata which are embedded into some JavaScript code running on the client-side. This code incrementally runs the field input strings on the automata. The effect is that each input field is associated with a regular language and at any point of time is decorated with one of four annotations: *green light* if the current value is a member of the language, meaning that it may be submitted; *yellow light* if the current value is a prefix of a member of the language, meaning that “the user is on the right way”; *n/a*, if the language is empty, which is used to denote that the field is not to be filled in at all; or *red light* if none of the above conditions apply, that is, the value is invalid and does not lead to something valid. The default behavior is that tiny icons inspired by traffic lights are placed beside the fields, denoting the annotations. Other icons can be chosen, such as checkmark symbols, arrows, etc. A form cannot be submitted if there is a yellow or red light; in that case, an error message specified by the programmer is shown instead.

The regular expressions, called *formats*, are defined in an XML syntax. Lots of syntactic sugar is offered, including standard UNIX regexp notation. Furthermore, expressions can be labeled for reference and reuse, and there is an import mechanism for modularization. The form field is associated with a format using a declaration as shown in the following example:

```
<form action="...">
  ISBN number:
  <input name="isbn">
  <input type="submit">
</form>
```

An HTML form

```
<format name="isbn"
  help="Enter an ISBN number"
  error="Illegal value entered">
  expression defining ISBN numbers
</format>
```

A PowerForms format declaration

Here the input field named `isbn` is associated with a help message, an error message, and a regular language. The example illustrates that the validation expressions can be completely separated from the HTML code—thereby the HTML code can be constructed independently of PowerForms and validation formats added later. Formats can be associated to all kinds of form fields, not just those of type `text`. For `select` fields, the format is used to filter the available options. For `radio` and `checkbox` fields, only the permitted buttons can be selected.

As noted in [79], many forms contain fields whose values may be constrained by the values entered in other fields. A typical example is fields that are not applicable if some other field has a certain value. Such interdependencies are almost always handled on the server, even if the rest of the validation is performed on the client-side. Presumably, the reason is that interdependencies require even more delicate JavaScript programming.

In PowerForms, interdependencies can be specified using an extension of the regular expressions: the `format` tags generally describe boolean decision trees, whose conditions probe the values of other form fields and whose leaves are simple formats. The interdependence is resolved by an iterative fixed-point process computed on the client by automatically generated JavaScript code.

As an extra feature, validating forms with field interdependencies can have side-

effects: if editing one field changes the format of another field, that field is “filtered”. For `select` fields, options that are no longer allowed are removed, and `radio` and `checkbox` buttons are released if their statuses become illegal. Since the side-effects only “remove” user data, they are monotonic and cyclic dependencies are resolved by the fixed-point process.

3.5.2 PowerForms in JWIG

PowerForms are available in the **JWIG** language through a variant of the `show` statement:

```
show h powerforms p;
```

where *h* is the XHTML document to be shown and *p* is a PowerForms document defining the input field validity requirements. Both *h* and *p* are values of type XML. Also the PowerForms document can be constructed dynamically using plug operations, which is often convenient. In fact, a variant of the XHTML 1.0 validity analysis described in Section 3.4.5 is applied here to check that the *p* document is always a valid PowerForms document according to a DSD2 description of the PowerForms language. This is achieved quite easily since the analysis algorithm is parameterized by the DSD2 description, and the syntax of PowerForms is easily expressed in DSD2.

The PowerForms formats can be coupled with the string analysis described in Section 3.4.2. Currently, we assume the worst case at each `receive` construct by modeling it as the total language. However, if a certain regular expression has been bound to a given input field using PowerForms, then PowerForms will guarantee that only such values can be received. This allows us to use that regular expression when analyzing the corresponding `receive` constructs to inexpensively but efficiently improve the analysis precision.

3.6 Other Aspects of Web Service Languages

There are many more sub-domains of the domain of interactive Web services. These include management of simultaneously executing session threads and a wide range of security issues. We describe these in the following, together with a notion of syntax macros, which is not tied to Web services in particular but primarily serves to connect the sublanguages of <bigwig>. More ideas and special features of <bigwig> and **JWIG** can be found in [40, 57].

3.6.1 Concurrency Control

With a number of session or interaction processes running concurrently, each interacting with a client and perhaps accessing shared data, there is an obvious need for concurrency control. A typical concurrency constraint will ensure exclusive access to a shared resource, for instance using the reader–writer protocol. A more complex concurrency constraint could informally sound like: “don’t allow a session to do *A* if *B* has occurred and *C* hasn’t occurred since that—unless *D* at some point has occurred,

or ...”. Large Web services quickly require quite complicated control which is hard to implement and maintain if using semaphores or other low-level primitives.

The <bigwig> approach is to use a high-level declarative logic-based language for specification of behavioral constraints, thereby increasing both succinctness and flexibility. The compiler translates these constraints into an operational specification, which at runtime is used to enforce a conforming global service behavior by a centralized controller component running on the server.

As mentioned in Chapter 2.1, M2L(Str), Monadic Second-order Logic on Finite Strings, can be used as a powerful temporal logic for specification of safety properties of distributed systems. The <bigwig> language contains a sublanguage where concurrency constraints can be specified essentially as M2L(Str) formulas. Using the MONA tool, these formulas are translated into minimal deterministic prefix-closed finite-state automata. These automata then act together as a *controller* at runtime, ensuring that all constraints are obeyed by delaying the sessions appropriately. An event is defined to be *enabled* if taking the corresponding step in the controller automata will stay within the accepting states.

As an example, the following constraint formula is used to implement an asynchronous event handler:

```
all t1: handle(t1) => is t2: t2<t1 && cause(t2) &&
                    (all t3: t2<t3 && t3<t1 => !handle(t3));
```

If `handle` occurs at time `t1`, then at some earlier time `t2` the event `cause` has occurred, and between `t2` and `t1` there has been no other `handle` event. In other words, this formula allows the handler to proceed, without blocking the application, whenever the associated event has been caused at least once since the last invocation of the handler.

The *events* on which the logic operates are in <bigwig> defined by “checkpoints”, each having the form of a `wait` statement:

```
wait {
  case A: ...
  case B: ...
  ...
  timeout t: ...
}
```

Executing such a `wait` statement causes the session to wait until the controller has determined that one of the events *A*, *B*, etc., is enabled. When permission to continue has been granted, the event is considered occurred and execution of the session is resumed at the corresponding location in the code. If a `timeout t` has been specified, the session waits at most *t* seconds. If the controller has not granted permission to continue within that time, the waiting is canceled and the session resumes execution at the code following the timeout specification.

Naturally, we do not expect common programmers to be willing to learn monadic second-order logic. Here the macro mechanism described in Section 3.6.3 comes in handy. It allows complex concurrency constraints to be wrapped into more user-friendly syntax.

More details of the <bigwig> concurrency control mechanism, including examples and a more detailed description of the implementation, can be found in [194, 40,

38]. There, it is also described how the technique is extended to constraints beyond those defining regular languages.

In the **JWIG** language, we do currently not apply temporal-logic based concurrency control. Instead, we rely entirely on the simpler standard Java features, such as synchronized blocks and monitor notification. This has so far appeared sufficient, and although the approach in <bigwig> has functioned as a successful proof of concept, we doubt that the benefits of introducing temporal logic to the **JWIG** programmers is worth the extra complexity of the language.

3.6.2 Security Issues

Typically operating on the global Internet, there are many aspects of Web service security. Communication can be eavesdropped, clients or servers are not always who they claim to be, services may be flooded with phony requests denying access for the real users, and seemingly indifferent programming errors may unintentionally open for sensitive data to be revealed from the server. In general, it may be of importance to ensure integrity of the service, authenticity of the server and client, and confidentiality of the communication.

The runtime system of <bigwig> and **JWIG** provides some generally applicable security arrangements. Integrity of the session thread state is achieved by keeping it exclusively on the server. Integrity of shared data can be provided by an underlying database. An interaction key is generated and embedded in every document shown to the client and serves to prevent submission of old forms. Clients and session threads are associated through a random key, which is created by the server upon the first request and carried across interactions in a hidden input field. In **JWIG**, a security manager is installed such that all installed services run in a secure sandbox on the server where they can only harm themselves. These basic mechanisms may optionally be combined with other protocols: SSL, the Secure Sockets Layer, for ensuring privacy of the communication and authentication of the server, and HTTP Authentication for ensuring authenticity of the client. Furthermore, the service can be configured to restrict access to certain IP addresses. This is useful for services that operate on the Internet but are intended for local use only. All security mechanisms mentioned here can be controlled via the high-level service API. In comparison, most other Web service languages provide similar control over SSL and HTTP Authentication, and we do not claim that <bigwig> or **JWIG** has any significant advantages here.

In addition to these mechanisms, we envision performing some simple static analyses relating to the behavioral security of particular services. Values are classified as *secret* or *trusted*, and, in contrast to tainting in Perl, the compiler keeps track of the propagation of these properties. Form data is always assumed to be untrusted, and secret values are never allowed to be plugged into gaps. The `system` function in <bigwig>, or equivalently, the `exec` method in **JWIG**, can only be called with a trusted string value. Variables can be declared with the modifiers *secret* or *trusted* and may then only contain the corresponding values. To change the classification of a value, there are two functions, `trust` and `disclose`. The programmer must make the explicit choice of using these coercions. The resulting mechanism is essentially an application of type qualifiers [89]. Such security systems have in other settings proven useful in preventing disclosure of sensitive information.

3.6.3 Language Abstractions with Syntax Macros

A central aspect of the <bigwig> language is its *macro mechanism*. This aspect does not have anything directly to do with Web service development; its purpose is to tie together the various sublanguages in <bigwig>, allowing convenient layers of language abstractions to be built on top. An alternative approach for making abstractions would be an object-oriented mechanism. However, due to the many different domain-specific sublanguages with highly different syntax, that would presumably become too restrictive.

It is often argued that macro mechanisms are low-level. A well-known example is C macros. They operate on the lexical level, meaning that they are unaware of the full language syntax. The macro mechanism employed in <bigwig> is a variant which operates on the *syntax level*. Our experience is that this approach is strong enough to support the construction of macros defined in terms of other macros, such that the macros form safe layers of abstractions, thereby increasing the stability of the resulting code, supporting code reuse, and decreasing development time.

Syntax-level macros operate on parse trees instead of token sequences. A macro has a “return type”, a nonterminal from the language grammar which the macro body must comply to. This allows syntactic discrepancies to be detected and reported properly. The syntax of macros may be chosen almost arbitrarily, making the mechanism flexible and transparent. Defining macros then feels like dynamically extending the language grammar with new productions.

In [40], a number of macro examples are given. They make it evident that syntax-level macros can raise the level of abstraction in the sublanguages for concurrency control, database support, form input validation, cryptographic security, etc. Also, it is shown how syntax macros can be taken to an extreme where they define whole new *very* domain-specific languages.

A macro mechanism resembling that of <bigwig> is described in [219]. The <bigwig> macros were first described in [34] and more recently in [41]. Work in progress by Brabrand and Schwartzbach aims for generalizing the mechanism, for instance, to make it possible to describe static and dynamic requirements on the use of macros and define more advanced syntactic transformations. This new edition is based on Java and will be used in **JWIG**, both for handling the special **JWIG** language constructs and for allowing also the **JWIG** application programmers to create and use syntax macros.

3.7 Schema Languages for XML

As explained in Section 3.3, XML plays an important role in the **JWIG** language and also in many other aspects of Web services. XML (Extensible Markup Language) [45] is a syntax derived from SGML for markup of text. XML is particularly interesting to computer scientists because the markup notation is really nothing but a way of specifying labeled trees. The tree view and the convenient SGML syntax of HTML have been important to the development of the World Wide Web, and, since its introduction, XML syntax has been hyped as a universal solution to the pervasive problem of format incompatibility for data exchange. We here consider one facet of XML: formal specification of the syntax of languages that use the XML notation. Other essential facets

include tree transformations, database-like querying, hyperlinking and addressing, and programming language APIs.

A *schema* is a description of the formal syntax of an XML language, in other words, it defines a set of XML trees. As for other formal languages, precise syntax descriptions provide an essential basis both for the tool builders and the application users. A *schema language* is a formal language for expressing schemas.

An XML document consists of named *elements* representing tree nodes. Elements have *attributes*, representing name/value pairs, and *content*, which is Unicode text called *chardata*, interspersed with subelements. We here ignore comments and DTD information, and we assume that entity references have been expanded. For example, consider the following XHTML document fragment:

```
<body class="mystuff">
  Hello <em>there</em>
</body>
```

This fragment contains an element named `body` that corresponds to a tree node labeled `body`. The node has an attribute named `class` and two children corresponding to its content, that is, the part between the start tag `<body...>` and the end tag `</body>`. The first child is a text node with value `Hello`, and the other is an element node labeled `em`. The `em` node in turn has one child node, which is a text node. XML markup is required to be *well-formed*, meaning that the begin and end tags are balanced and nested properly, which allows us to view XML documents as tree structures. More details about the XML data model and a general overview of other aspects of XML can be found in [162].

A schema for XHTML would for example state that `class` attributes in fact are allowed in `body` elements, that `chardata` is allowed in the content, but also that, for instance, `body` elements cannot appear within the `em` tags. A schema language should make it possible to easily express such constraints.

Numerous schema language proposals have already emerged. Among these are DTD [45], DDML [32], DCD [43], XML-Data [145], XDR [92], SOX [70], TREX [60], Schematron [119], Assertion Grammars [184], and RELAX [169]. W3C has issued their XML Schema proposal [209] in an attempt to reconcile the efforts. However, it has been met with intense debate, primarily due to its unprecedented complexity viewed by many as being unnecessary and harmful [191, 2]. Concurrently, RELAX NG [62] has been developed as a descendant of RELAX and TREX and is now being standardized by OASIS and ISO. The many proposals, and the outcome of the XML Schema effort, indicate that it is far from obvious how the right schema language should be designed. In general, the XML notation turns out to be so versatile that it is hard to satisfy all design requirements and capture the various usage patterns, and at the same time keep the schema notation simple.

Our *Document Structure Description* (DSD) proposal has the ambition of providing an expressive power comparable to that of the most significant alternatives—XML Schema and RELAX NG—while at the same time remaining simple to learn and use. In [133], we thoroughly compare DSD with DTD, XML Schema, and RELAX NG.

We have tried to identify the most central aspects of XML language syntax and turn these into a clean set of schema constructs, based on well-known computer science concepts, such as boolean logic and regular expressions. A DSD schema defines a

grammar for a class of XML documents, including documentation for that class, and additionally a method for specifying default parts of documents. As most other schema language proposals, the DSD language itself uses the XML notation. This opens up for the possibility of being self-describing, that is, having a DSD description of the DSD language.

In the following sections, we present a brief overview of DSD and its successor, DSD2, and present the algorithm for checking whether all XML documents represented by a given summary graph are within the language specified by a given DSD2 schema, which we use in the analysis of **JWIG** programs, as mentioned in Section 3.4.5.

3.7.1 The Document Structure Description Language

A DSD, that is, a document written in the DSD language, defines the syntax of a family of conforming XML documents. An *application document*, also called an *instance document*, is an XML document intended to conform to a given DSD. It is the job of a *DSD processor* to determine whether or not an application document is conforming. This section describes the main aspects of the DSD language. A more thorough description can be found in [133]; for the rigorous definition, see the language specification [132].

To give an impression of the DSD language, we first show a fragment of a small example DSD for “business cards”:

```
<DSD IDRef="card" DSDVersion="1.0">
  <Title> This is a DSD for XML business cards </Title>
  <ElementDef ID="card">
    <AttributeDecl Name="type" Optional="yes">
      <Union><String Value="simple"/><String Value="complex"/></Union>
    </AttributeDecl>
    <Element Name="name"><StringType/></Element>
    <If><Attribute Name="type" Value="simple"/>
      <Then><Element Name="title"><StringType/></Element>
      ...
    </Then>
    </If>
    ...
  </ElementDef>
  ...
</DSD>
```

and a conforming application document:

```
<card type="simple">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 456-1414</phone>
</card>
```

The DSD specifies the format of a business card as a `card` root element, which contains an optional `type` attribute with value `simple` or `complex`, representing two variants of

cards. Both variants have a `name` subelement containing pure chardata, whereas, in the fragment shown here, only “simple” cards have a `title` subelement.

One advantage of XML is that anybody can create new XML-based languages like this and freely benefit from the huge supporting framework of tools and technologies that surround the basic notation of XML, for instance schema processors. In particular, having a schema description of the language syntax provides a formal but human readable reference, and specialized tools using the language can apply preexisting schema processors to validate their input, which simplifies implementation.

A DSD processor basically performs one top-down traversal of the application document tree in order to check conformance. During this traversal, constraints and other requirements from the DSD are *evaluated* relative to a *current element* of the application document. The DSD processor consults the DSD to determine the constraints that are *assigned* to each node for later evaluation. Initially, a constraint is assigned to the root node. Evaluation of a constraint may entail the insertion of default attribute values and default content in the current element. Also, it may assign constraints to the subelements of the current element. If no constraints are violated during the entire tree traversal, the original document conforms to the DSD. The document augmented with inserted defaults constitutes the result of the DSD processing.

A DSD consists of a number of definitions, each associated with an ID for cross-referencing. In the following, the various kinds of DSD definitions and other features are briefly described:

Element constraints The central definition in DSD is the *element definition*. An element definition specifies an element name and a *constraint*. During conformance checking, each element node in the application document is assigned an ID referring an element definition from the DSD. In order for the element node to match the element definition, they must have the same name, and the element node must satisfy the constraint.

A constraint is defined by a number of constraint expressions, which can contain declarations of attributes and element content, boolean expressions about attributes and context, and conditional subconstraints. The constraint is satisfied if the evaluation of each constituent succeeds. Boolean expressions are built from the usual boolean operators and are used for several purposes: they express dependencies between attributes, and they are used as guards in conditional constraints and default declarations, as explained later.

Attribute declarations During evaluation of a constraint, attributes are *declared* gradually. Only attributes that have been declared are allowed in an element. Since constraints can be conditional and attributes are declared inside constraints, this evaluation scheme allows hierarchical structures of attributes to be defined. Such structures cannot be described by other schema proposals, although they commonly appear in XML languages. Schemas written in other languages are therefore often too loose, meaning that they allow more documents as being valid than intended. Concrete examples of this are given in [162, 133].

An *attribute declaration* consists of a name and a string type. The name specifies the name of the attribute, and the string type specifies the set of its allowed values. Unless it is declared as optional, an attribute must be present if it is

declared. The presence and values of declared attributes can be tested in boolean expressions and in context patterns, which are described below.

String types A *string type* is a set of strings defined by a regular expression. String types are used for two purposes: to define valid attribute values and to define valid chardata. Regular expressions provide a simple, well-known, and expressive formalism for specification of sets of strings—as we have seen previously, for instance in the PowerForms language in Section 3.5. As in PowerForms, we here include a number of nonstandard operators, such as, intersection and complement.

Content expressions Recall that the content of the current element is a sequence of element nodes and chardata nodes. *Content expressions* are used to specify sets of such sequences using a kind of regular expressions. The atomic expressions are either element description or string types, which match elements and chardata nodes, respectively. Checking that content sequences satisfy the given constraints has the side-effect that element definition IDs are assigned to the subelements for continuing the top-down processing. Also, as explained below, insertion of default content occurs while checking content expressions. Because of these side-effects, we use a non-standard operational interpretation of the regular expression constructs occurring in content expressions, in order to get a well-defined behavior.

If more than one content expression is specified, each of them must match some of the content of the current element, just like each attribute declaration must match an attribute. More precisely, each content expression is matched against a subsequence of the content that consists of elements mentioned in the content expression itself. This method makes it easy to combine requirements of both ordered and unordered content, and additionally, unordered content is declared just like attributes.

Context patterns A *context pattern* can be used with defaults, constraints and content descriptions to make them context dependent. As the conditional declarations mentioned above, context dependency is a phenomenon that other schema languages cannot express directly, although it is common in XML languages. For instance, in XHTML, a elements cannot be nested, but this requirement is not expressible in either DTD or XML Schema, and in RELAX NG only by effectively doubling the schema size for each such constraint. A context pattern is a restricted form of regular expression that looks at the list of ancestor elements of the current element.

Default insertion It is convenient to application document authors to be able to omit implied attributes and other document parts. Since schemas describe the document structure, they are a suitable place to specify default values for such parts. Validating a document then has the side-effect of inserting the defaults, which is often useful to subsequent document processing.

In DSD, default attributes and contents are defined by an association to a boolean expression. Such attributes or contents are applicable for insertion at a given place in the application document if the boolean expression evaluates to true at

that place. An attribute default can be inserted when an attribute declaration is encountered and the declared attribute is not present in the current element. Similarly, during evaluation of a content expression, if an element description or a string type is encountered and the next content node does not match the description, an applicable default can be inserted.

In addition to the main features mentioned above, DSD also contains notions of ID attributes and points-to requirements for describing intra-document references, it allows structured self-documentation to be embedded within schemas, and it supports modularity and reuse of descriptions by document inclusion and selective redefinitions. Furthermore, the DSD language is self-describable: There is a *meta-DSD*—a DSD description of DSD, which is complete, in the sense that an XML document is syntactically a DSD schema if and only if it is valid relative to this meta-DSD. This property of being entirely self-describable is not only aesthetically pleasing, it is also practically useful for application development, as noted in [133].

By the many proposals for XML schema languages, it appears that there is no ideal solution to the problem of designing the right schema language that fits all purposes. Still, we believe that the DSD language has succeeded in identifying the most central aspects of defining sets of XML document, and that it provides a simple but expressive alternative to other proposals. Specifically, we believe that in particular the following ideas have proven successful: the application of context expressions, boolean logic, and conditional constraints to describe dependencies, the flexible content model, the declarative default mechanism, and the explicit top-down traversal method.

3.7.2 Validating Summary Graphs with DSD2

Based on experience with DSD and other recent schema languages, a successor, DSD2, has been developed, as described in [57]. A normative specification document for DSD2 is currently under development [160]. The goal for the design of DSD2 has been to create a schema language that is even simpler than DSD and at the same time also more expressive.

Conceptually, a DSD2 schema consists of a list of *constraints*. A constraint is either a *declaration*, a *requirement*, a *conditional constraint*, or a *default specification*. Furthermore, there are notions of *string normalization*, *keys* and *references*, and *options* which we ignore here. The main differences from DSD are: 1) the notion of conditional constraints has been strengthened, since a schema is now essentially defined by a collection of such constructs; 2) the idea of assigning element definition IDs to the application document elements has been replaced by more directly referencing to elements using their names; and 3) content expressions are now normal regular expressions with a standard semantics, as opposed to the operational variant used in DSD.

During a top-down traversal of the application document, the processor checks each element in turn, as in DSD. This check of an individual element is performed in five steps:

1. all applicable constraints are found;
2. default attributes and contents are inserted;

3. the requirements are checked;
4. it is checked that all attributes of the current element are declared; and
5. it is checked that the contents matches all content declarations and that the whole content is declared.

The following description of the various types of constraints explains these steps in more detail:

Declarations A declaration constraint contains a list of *attribute declarations* and *content declarations*. An attribute declaration specifies that an attribute with a given name is allowed in the current element provided that the value matches a given regular expression. A content declaration is a regular expression over characters and element names that specifies a requirement for the presence, ordering, and number of occurrences of subelements and character data. As in DSD, a content declaration only looks at elements that are mentioned in the regular expression. All attributes and contents that have been matched by a declaration are considered to be *declared*.

Requirements A requirement constraint contains boolean expressions that must evaluate to true for the current element. Boolean expressions are built of the usual boolean operators, together with attribute expressions probing the presence and values of attributes, and element expressions probing the name of the current element. The context expressions from DSD have been replaced by simpler *parent* and *ancestor* operators which probe whether certain properties are satisfied for the elements above the current element in the instance document tree.

Conditional constraints A conditional constraint contains a list of subconstraints whose applicability is guarded by a boolean expression. Only when the boolean expression evaluates to true for the current element, the subconstraints are processed.

Defaults A default constraint specifies a default value for an attribute or a default contents sequence for an empty element. This restriction of only allowing default contents for empty elements, instead of being able to insert individual elements within existing contents as in DSD, together with the fact that there is no longer any notion of assigning element definition IDs to subelements allows us to use a standard regular expression semantics for the content expressions.

Additionally, specifications can be grouped and named for modularity and reuse, and, in contrast to DSD, Namespaces [44] are now fully supported.

An Algorithm for Summary Graphs Validation

The DSD2 processing algorithm explained above generalizes from concrete XML documents to the concept of summary graphs described in Section 3.4.2. The overall benefit of using DSD2 instead of, for instance, DTD or XML Schema is that it allows us to precisely and efficiently capture more programming errors, yet with a reasonably straightforward validation algorithm.

Given a DSD2 schema and a summary graph associated to some `show` statement, we must check that every XML document in the language of the summary graph is valid according to the schema. For validation in **JWIG**, we use a DSD2 schema for XHTML 1.0, specifically. Our algorithm for validating a summary graph with respect to a DSD2 schema proceeds in a top-down manner mimicking the definition of the unfolding relation described in Section 3.4.2, starting from the root elements in the templates of the summary graph root nodes. The constraints are then evaluated symbolically on all graph unfoldings. For each element, essentially the same five steps mentioned above are performed, but now taking the gaps and edges into consideration. Using a standard memoization technique, we ensure termination, even in case of loops in the summary graph.

In this symbolic evaluation, especially content expressions and boolean expressions require special attention. Evaluation of content expressions is complicated by the presence of gaps in the content sequences. The template edges from the gaps may lead to templates which at the top-level themselves contain gaps. In turn, this may cause loops of template edges. Therefore, in general, the set of possible contents sequences of a given element forms a context-free language, which we can represent by a context-free grammar. The problem of deciding inclusion of a context-free language in a regular language is decidable [108], but computationally expensive. For that reason, we approximate the context-free language by a regular one and apply a simpler regular language inclusion algorithm. Although loops in summary graph often occur, our experiments show that it is rare that this approximation actually causes any imprecision.

Also for boolean expressions in conditional constraints and requirement constraints, evaluation is nontrivial because we simultaneously consider all the possible unfoldings of the summary graph. We apply a *four-valued* logic for that purpose. Evaluating a boolean expression results in one of the following values:

- true* – if evaluating the expression on every possible unfolding would result in *true*;
- false* – if they all would result in *false*;
- some* – if some unfolding would result in *true* and others in *false*;
- don't-know* – if the processor is unable to detect whether all, no, or some unfoldings would result in *true*.

All boolean operators extend naturally to these values. The value *don't-know* is for instance produced by the conjunction of *some* and *some*. If the guard of a conditional constraint evaluates to *don't-know*, we terminate with a “don't know” message. However, for our concrete XHTML schema, this can in fact never happen.

The validation algorithm is described in more detail in [57]. The algorithm is sound, that is, if it validates a given summary graph, it is certain that all unfoldings into concrete XML documents are also valid. It is generally not complete, but no false errors have been encountered for our **JWIG** benchmark programs. Furthermore, for a fixed schema, for instance the one for XHTML used in **JWIG**, the algorithm runs in linear time in the size of the templates appearing in the given summary graph, which can be considered optimal.

Chapter 4

Conclusion

We have in Chapter 2 presented an overview of MONA—an efficient implementation of an automata-based decision procedure for the logic WS1S and a number of related logics. (Chapters 5–8 contain papers that go into more detail.) This implementation has provided the foundation of or been integrated into a number of other tools. More than 25 publications describe tools or techniques that use MONA. We have focused on PALE, which is a program verifier for pointer-intensive programs. As part of this verification technique, we have introduced a formal language, Pointer Assertion Logic (PAL), for specifying properties of data structures. This language is designed to maximize the potential of the verification technique with respect to the underlying MONA decision procedure. The WS1S logic and the PAL language essentially correspond to the class of regular languages over finite strings or trees. In that perspective, both MONA and PALE are extraordinary examples of the expressive power and versatile nature of the regular languages.

In Chapter 3, we have shown a number of languages together with implementation and analysis methods that are designed to improve development of Web services. (The papers in Chapters 9–16 contain more details of this topic.) The `<bigwig>` and **JWIG** languages are high-level programming languages especially designed for development of interactive Web services. While `<bigwig>` is designed from scratch with inspiration from existing languages, **JWIG** is a variant of Java extended with special syntactic constructs and program analyses. Common to these languages is a session-centered runtime model, the DynDoc sublanguage for flexible construction of Web pages, and PowerForms for declarative specification of form input validation. Additionally, the `<bigwig>` language contains a sublanguage for concurrency control based on a variant of WS1S and a macro mechanism working on the syntactic language level.

The DSD and DSD2 languages are developed to be able to concisely describe common syntactic requirements in XML-based languages. The XML notation is being used extensively for data representation in many different application domains, whereof one is Web services. In particular, the syntax of Web pages can be defined precisely in DSD and DSD2. We use this property to construct program analyses for `<bigwig>` and **JWIG** to statically ensure that only syntactically correct Web pages can be shown, without limiting the flexibility in DynDoc. Related to this, we introduce specialized program analyses to check consistency of the dynamic construction of Web pages and of the use of input forms in the pages. A fundamental aspect of the

program analyses in **JWIG** is the notion of summary graphs for abstractly describing how the Web pages are being constructed in a given program.

In these chapters we have seen a diverse collection of domain-specific formal languages. First, we studied the domain of regular sets of finite strings or trees and its use in program verification in the MONA and PALE projects; in the second part, we looked into aspects of the domain of Web services in the <bigwig>, **JWIG**, and DSD projects. We have applied the domain-specific language (DSL) paradigm in various ways in all projects, either during the language design or implementation phases or when devising analysis or verification techniques. In this sense, the projects have overall commonalities in spite of the diversity.

In the MONA project, our focus has been on creating an efficient implementation. WS1S and the related logics have long been known as exceptionally concise notations for regular languages, but it was not known before the MONA tool was built that their decision procedures could be implemented effectively. The goal of the PALE project has been to design a language for expressing properties of data-type implementations, such that efficient verification with the MONA tool is feasible, while at the same time providing as much expressive power as possible. Thus, this project has involved a combination of language design, implementation considerations, and verification technique development. The underlying DSL paradigm has helped in creating solutions that are both simple and powerful.

In the <bigwig> project, the domain of Web services was initially analyzed to identify the most essential aspects that could benefit from high-level language support. Based on this analysis, we focus on a few key ideas, such as, the session-centered model, Web page construction using higher-order templates, and regular languages for validating form input. The explicit language-based mechanisms make it possible to perform domain-specific program analyses. Due to the DSL approach, <bigwig> and the successor **JWIG** are, for example, the only existing programming languages for Web service development that provide static guarantees for syntactic correctness of the constructed Web pages without sacrificing flexibility of the Web page construction mechanism.

The development of DSD was initially motivated by the lack of schema languages for XML that are simple to learn and use and also powerful enough to allow most common syntactic requirements to be expressed. In parallel with the development of the DSD and DSD2 schema languages, other alternatives have emerged with the same motivation, but we argue that DSD and DSD2 still provide significant benefits. Again, using the DSL approach has resulted in domain-specific languages whose constructs closely match what the user wants to express at high levels of abstraction.

We conclude that in each of these projects where a formal language has been created, the ideas in the domain-specific language paradigm have contributed to the achievements of the projects.

Part II

Publications

Chapter 5

MONA 1.x: New Techniques for WS1S and WS2S

with Jacob Elgaard and Nils Klarlund

Abstract

In this note, we present the first version of the MONA tool to be released in its entirety. The tool now offers decision procedures for both WS1S and WS2S and a completely rewritten front-end. Here, we present some of our techniques, which make calculations couched in WS1S run up to five times faster than with our pre-release tool based on M2L(Str). This suggests that WS1S—with its better semantic properties—is preferable to M2L(Str).

5.1 Introduction

It has been known for a couple of years that Monadic Second-order Logic interpreted relative to finite strings, M2L(Str), is an attractive formal and practical vehicle for a variety of verification problems. The formalism is generally easy to use, since it provides boolean connectives, first and second-order quantifiers and no syntactic restrictions, say, to clausal forms. However, the semantics of the formalism is the source of definitional and practical problems. For example, the concept of a first-order term doesn't even make sense for the empty string since such terms denote positions.

So, it is natural to investigate whether the related logic WS1S (Weak Second-order theory of 1 Successor) can be used instead. This logic is stronger in that it captures a fragment of arithmetic, and its decision procedure is very similar to that of M2L(Str). Similarly, we would like to explore the practical feasibility of WS2S (Weak Second-order theory of 2 Successors).

In this note, we present some new techniques that we have incorporated into the first full release of the MONA tool. The MONA tool consists of a front-end and two back-ends, one for WS1S and one for WS2S. The front-end parses the MONA program, which consists of predicates (subroutines that are compiled separately), macros, and a main formula. Each back-end implements the automata-theoretic operations that are carried out to decide the formula corresponding to the program.

Since our earlier presentation of the MONA tool [24], we have completely rewritten the front-end, this time in C++ (the earlier version was written in ML). In the old version, the front-end produces a *code tree*, whose internal nodes each describe an automata-theoretic operation—such as a product or subset construction—and whose leaves describe automata corresponding to basic formulas. We implemented optimization techniques (unpublished) based on rewriting of formulas according to logical laws. In this note, we report on an alternative optimization technique, based on building a code DAG instead of a code tree. (A DAG is a directed, acyclic graph.) Experiments show that this technique together with a more efficient handling of predicates yields up to five-fold improvements in compilation time over the old tool.

We also briefly discuss how a M2L(Str) formula can be translated into an essentially equivalent WS1S formula, and we discuss important problems to be addressed.

5.2 M2L(Str) and WS1S

M2L(Str)

A formula of the logic M2L(Str) is interpreted relative to a number $n \geq 0$, which is best thought of as defining the set of positions $\{0, \dots, n-1\}$ in a string of length n . The core logic consists of first-order terms, second-order terms, and formulas. A *first-order term* t is a variable p , a constant 0 (denoting the position 0, which is the first position in w) or \$ (denoting $n-1$, which is the last position in the string), or of the form $t' \oplus 1$ (denoting $i+1 \bmod n$ when t' is a first-order term denoting i). A *second-order term* is either a variable P or of the form $T' \cup T''$. A formula ϕ is either a basic formula of the form $t \in T$ or $T \subset T'$, or of the form $\psi \wedge \chi$, $\neg\psi$, $\exists p : \psi$ (first-order quantification), or $\exists P : \psi$ (second-order quantification). In addition, we allow formulas involving $=$ (between first-order or second-order terms); $<, \leq, >, \geq$ (between first-order terms); boolean connectives $\Rightarrow, \Leftrightarrow$ and \vee ; set operations \cap, \setminus , and \mathbb{C} ; \forall quantifiers; etc.

The automaton-logic connection (see [126]) allows us to associate a regular language over \mathbb{B}^k , for some $k \geq 0$, to each formula ϕ as follows. We assume that there are k variables that are ordered and that include the free variables in ϕ . Now, a string w of length n over the alphabet \mathbb{B}^k can be viewed as consisting of k *tracks* (or rows), each of length n . The k th track is a bit-pattern that defines the interpretation of the k th variable, assumed to be second-order, as the set of positions i for which the i th bit is 1. Note that a first-order variable can be regarded as a second-order variable restricted to singleton values, so the assumption just made that variables are second-order is not a serious one. The *language* associated with formula ϕ is now the set of all strings that correspond to a satisfying interpretation of the formula. As an example, the formula $P \subseteq Q$ is associated with the regular language

$$((\binom{0}{0}) + (\binom{0}{1}) + (\binom{1}{1}))^*$$

where the upper track of a string denotes the value of P and the lower track denotes the value of Q . Any language corresponding to a formula is regular, since the languages corresponding to basic formula can be represented by automata, and \wedge , \neg , and \exists correspond to the automata-theoretic operations of product, complementation, and projection. In the case of a closed formula with no free variables, the regular language

degenerates to a set of strings over a unit alphabet. Thus a closed formula essentially denotes a set of numbers.

The proof of regularity just hinted at forms the basis for the decision procedure: each subformula is compiled into a minimum deterministic automaton, see [126]. An automaton representation based on BDDs is at the core of the MONA implementation as discussed in [126]. For each state p in the state space S , a multi-terminal BDD whose leaves are states represents the transition function $a \mapsto \delta(p, a) : \mathbb{B}^k \rightarrow S$ out of p . Each BDD variable corresponds to a first or second-order WS1S variable, and the BDDs are shared among the states. Thus the resulting data structure is a DAG with multiple sources.

The automaton-logic connection (see [126]) allows us to associate a regular language over \mathbb{B}^k to each formula ϕ that has k variables.

WS1S

WS1S has the same syntax as M2L(Str) except that there is no \complement operator and $\oplus 1$ is replaced with $+1$. This logic is interpreted in a simpler manner: first-order terms denote natural numbers, and second-order terms denote finite sets of numbers. The automata-theoretic calculations are similar to that of M2L(Str) except for the existential quantifier (see [126]).

From M2L(Str) to WS1S

In principle, it is easy to translate a quantifier free M2L(Str) formula ϕ to a formula ϕ' in WS1S with essentially the same meaning: ϕ' is gotten from ϕ by the following steps.

- A conjunct $p \leq \$$, where $\$$ now is a variable, must be added to any subformula of ϕ containing a first-order variable p .
- Each second-order variable P is left untouched, so that the translated formula will not depend on whether P has any elements greater than $\$$. However, occurrences of \emptyset must be taken into account; for example, the formula $P = \emptyset$ is translated into $\forall p \leq \$: \neg(p \in P)$ so that the translated formula does not depend on the membership status of numbers in P that are greater than $\$$. Any use of set complement operator \complement must also be carefully replaced.
- Any occurrence of a subformula involving \oplus such as $p = q \oplus 1$ must be replaced by something that captures the modulo semantics (here: $q < \$ \Rightarrow p = q + 1 \wedge p = \$ \Rightarrow p = 0$).

With such a scheme it can be shown that I for length $n > 0$ satisfies ϕ if and only if I , augmented by interpreting $\$$ as $n - 1$, satisfies ϕ' . Unfortunately, in order to preserve this property for all subformulas, we need to conjoin extraneous conditions onto every original subformula. A simpler solution is to conjoin them only for certain strategic places, such as for all basic formulas and all formulas that are directly under

a quantifier. We have implemented such heuristics in a tool, `s2N`, that automatically translates M2L(Str) formulas to WS1S formulas.¹

5.3 DAGs for Compilation

Code trees can be of the form (among others) `mk-basic-less(i, j)`, `mk-product(C, C', op)`, or `mk-project(i, C)`, where i and j are BDD variable indices, op is a boolean function of two variables, and C and C' are code trees. For example, consider the formula $\exists q : p < q \wedge q < r$. If variable p has index 1, i.e., if it is the 1st variable in the variable ordering, variable q has index 2, and variable r has index 3, then this formula is parsed into a code tree `mk-project(2, mk-product(mk-less(1, 2), mk-less(2, 3), \wedge))`. This tree contains a situation that we would like to avoid: essentially isomorphic subtrees are processed more than once. In fact, the automaton A for `mk-less(1, 2)` is identical to the automaton A' for `mk-less(2, 3)` modulo a renaming of variables. In general, we would like to rename the indices in A whenever we need A' , since this is a linear operation (whereas building A or A' from the code tree is often not a linear operation).

So, we say that a code tree C is *equivalent* to C' if there is an order-preserving (i.e., increasing), renaming of variables in C' such that C' becomes C . Our goal is to produce the DAG that arises naturally from the code tree by collapsing equivalent subtrees. Unfortunately, it takes linear time to calculate the equivalence class of any subtree, and so the total running time becomes quadratic. Therefore, the collapsing process is limited to subtrees for which the number of variable occurrences is less than a user definable parameter ℓ .²

MONA offers both pre-compiled subroutines, called *predicates*, and typed macros. A use `name(\vec{X})` of a predicate, where \vec{X} is a sequence of actual parameters, is translated to a special node of the form `mk-call(name, \vec{X})`. The predicate is then compiled separately given the signature of the call node. The actual parameters are bound to the resulting automaton using a standard binding mechanism: introduction of temporary variables and projection. Additional call nodes with the same signature can then reuse the separately compiled automaton. Call nodes act as leaves with respect to DAGification.

5.4 Experimental Results

We have run a MONA formula, `reverse`, of size 50KB (an automatically generated formula from [120]) through our old MONA (using optimizations) and our new WS1S version with and without DAGification ($\ell = 200$). We also did the experiment on `reverse2`, a version of the formula where all defined predicates were replaced by

¹This approach has been made obsolete by the notions of restrictions with three-valued logic and automata that are described in Section 2.1.7 and in [128]. Experiments with the `s2N` tool provided the motivation for those ideas.

²Since MONA version 1.3, this limitation has been removed due to a more efficient formula representation. In all non-artificial examples, the time requirements for performing DAGification have since been negligible. The benefits of performing DAGification on the subsequent automata operations are measured in [134].

macros. And, we have run a comparison on a formula representing a parameterized hardware verification problem. The results are (in seconds):

Program	Old MONA	MONA 1.1	w. DAGs	DAG Hits	DAG Misses
reverse	17	8.5	3.0	20513	2725
reverse2	51	90	45	327328	14320
hardware	6.6	5.4	4.7	3284	633

In some cases (like in `reverse2`), the old MONA tool is faster than the new one run without DAGification, since the figures reported for the old apply to the version that carries out formula simplification. The experiments support our claim that WS1S can be as efficient a formalism as M2L(Str). (The underlying BDD-package in the two tools is the same.) Moreover, our DAGs and predicate uses offer substantial benefits, up to a factor five. The `hardware` example runs only slightly faster, and the improvement is due to the new front-end being quicker.

5.5 Related and Future Work

There are at least three similar tools reported in the literature: [98] reports on an implementation of WS1S that is not based on BDDs and that therefore is likely not to be as efficient as our tool. The tool in [123] implements M2L(Str) using a different BDD representation, and the tool in [165] implements a decision procedure for WS2S (in Prolog and without BDDs).

There are still several problems and challenges not addressed in the current MONA tool: 1) the semantics of formulas with first-order terms is not appealing, for example, the MONA formula $x_1 < x_2 \wedge \dots \wedge x_{n-1} < x_n$ is translated in linear time whereas its negation, $x_1 \geq x_2 \vee \dots \vee x_{n-1} \geq x_n$, is translated in exponential time; 2) there is no reuse of intermediate results from one automaton operation to the next (a general solution to this problem seems to require identification of isomorphic subgraphs, a problem that appears computationally expensive); 3) the automatic translation from M2L(Str) to WS1S by s2N sometimes makes formulas unrunnable for reasons similar to 1), namely that the restrictions a formula is translated under are wrapped into subformulas in unfortunate ways unless the restrictions are reapplied for each intermediate result; 4) the use of formula rewriting (as we did in the earlier MONA version) should be combined with our DAG techniques.³

The MONA tool, currently⁴ in version 1.2, can be retrieved from <http://www.brics.dk/mona>, along with further information.

³Much of this has changed in later versions of MONA. The first and the third problem have been solved by the use of three-valued logic and automata. The fourth issue has been addressed as explained in Section 2.1.8 and in [134, 131].

⁴as of June 1998

Chapter 6

MONA Implementation Secrets

with Nils Klarlund and Michael I. Schwartzbach

Abstract

The MONA tool provides an implementation of automaton-based decision procedures for the logics WS1S and WS2S. It has been used for numerous applications, and it is remarkably efficient in practice, even though it faces a theoretically non-elementary worst-case complexity. The implementation has matured over a period of six years. Compared to the first naive version, the present tool is faster by several orders of magnitude. This speedup is obtained from many different contributions working on all levels of the compilation and execution of formulas. We present an overview of MONA and a selection of implementation “secrets” that have been discovered and tested over the years, including formula reductions, DAGification, guided tree automata, three-valued logic, eager minimization, BDD-based automata representations, and cache-conscious data structures. We describe these techniques and quantify their respective effects by experimenting with separate versions of the MONA tool that in turn omit each of them.

6.1 Introduction

MONA [105, 131, 158, 126] is an implementation of decision procedures for the logics WS1S and WS2S (Weak monadic Second-order theory of 1 or 2 Successors) [208]. They have long been known to be decidable [49, 83], but with a non-elementary lower bound [156]. For many years it was assumed that this discouraging complexity precluded any useful implementations.

MONA has been developed at BRICS since 1994, when our initial attempt at automatic pointer analysis through automata calculations took four hours to complete. Today MONA has matured into an efficient and popular tool on which the same analysis is performed in a couple of seconds. Through those years, many different approaches have been tried out, and a good number of implementation “secrets” have been discovered. This paper describes the most important tricks we have learned, and it tries to quantify their relative merits on a number of benchmark formulas.

Of course, the resulting tool still has a non-elementary worst-case complexity. Perhaps surprisingly, this complexity also contributes to successful applications, since it is provably linked to the succinctness of the logics. If we want to describe a particular regular set, then a WS1S formula may be non-elementarily more succinct than a regular expression or a transition table.

The niche for MONA applications contains those structures that are too large and complicated to describe by other means, yet not so large as to require infeasible computations. Happily, many interesting projects fit into this niche, including hardware verification [18, 11], pointer analysis [120, 82, 161], controller synthesis [194, 113], natural languages [164], parsing tools [69], software design descriptions [130], Presburger arithmetic [198], and verification of concurrent systems [136, 135, 121, 180, 200].

There are a number of tools resembling MONA. Independent of the MONA project, the first implementation of automata represented with BDDs was that of Gupta and Fischer from 1993 [102]. However, they used “linear inductive functions” instead of the automaton–logic connection. MOSEL (see <http://sunshine.cs.uni-dortmund.de/projects/mosel/>) implements the automaton based decision procedure for M2L(Str) using BDDs like MONA. In [123], MOSEL is described and compared with MONA 0.2, which provided inspiration for the MOSEL project. Apparently, there have been only few applications of MOSEL. AMORE [153] (see <http://www.informatik.uni-kiel.de/inf/Thomas/amore.html>) is a library of automata theoretic algorithms, resembling those used in MONA. AMORE also provides functionality for regular expressions and monoids, but is not tied to the automaton–logic connection. Glenn and Gasarch [98] have in 1997—apparently independently of MONA and MOSEL—implemented a decision procedure for WS1S, basically as the one in MONA, but without using BDDs or other sophisticated techniques. The SHASTA tool from 1998 is based upon the same ideas as MONA. It is used as an engine for Presburger Arithmetic [198].

Furthermore, MONA has provided the foundation of or been integrated into a range of other tools: FIDO [140], LISA [12], DCVALID [180], FMONA [29], ST-TOOLS [187], PEN [175], PAX [19], PVS [178], and ISABELLE [16].

6.2 The Automaton–Logic Connection

Being a variation of first-order logic, WS1S is a formalism with quantifiers and boolean connectives. First-order terms denote natural numbers, which can be compared and subjected to addition with constants. Also, WS1S allows second-order terms, which are interpreted as finite sets of numbers. The actual MONA syntax is a rich notation with constructs such as set constants, predicates and macros, modulo operations, and let-bindings. If all such syntactic sugar is peeled off, the formulas are “flattened” (so that there are no nested terms), and first-order terms are encoded as second-order terms, the logic reduces to a simple “core” language:

$$\begin{array}{lcl} \varphi & ::= & \sim\varphi' \quad | \quad \varphi' \ \& \ \varphi'' \quad | \quad \text{ex2 } X_i : \varphi' \\ & & X_i \text{ sub } X_j \quad | \quad X_i = X_j \setminus X_k \quad | \quad X_i = X_j + 1 \end{array}$$

where X ranges over a set of second-order variables.

Given a fixed main formula φ_0 , we define its semantics inductively relative to a string w over the alphabet \mathbb{B}^k , where $\mathbb{B} = \{0, 1\}$ and k is the number of variables in φ_0 . We assume every variable of φ_0 is assigned a unique index in the range $1, 2, \dots, k$, and that X_i denotes the variable with index i . The projection of a string w onto the i 'th component is called the X_i -track of w . A string w determines an interpretation $w(X_i)$ of X_i defined as the finite set $\{j \mid \text{the } j\text{th bit in the } X_i\text{-track is } 1\}$.

The semantics of a formula φ in the core language can now be defined inductively relative to an interpretation w . We use the notation $w \models \varphi$ (which is read: w satisfies φ) if the interpretation defined by w makes φ true:

$$\begin{array}{ll}
 w \models \sim \varphi' & \text{iff } w \not\models \varphi' \\
 w \models \varphi' \ \& \ \varphi'' & \text{iff } w \models \varphi' \text{ and } w \models \varphi'' \\
 w \models \text{ex2 } X_i : \varphi' & \text{iff } \exists \text{ finite } M \subseteq \mathbb{N} : w[X_i \mapsto M] \models \varphi' \\
 w \models X_i \text{ sub } X_j & \text{iff } w(X_i) \subseteq w(X_j) \\
 w \models X_i = X_j \setminus X_k & \text{iff } w(X_i) = w(X_j) \setminus w(X_k) \\
 w \models X_i = X_j + 1 & \text{iff } w(X_i) = \{j + 1 \mid j \in w(X_j)\}
 \end{array}$$

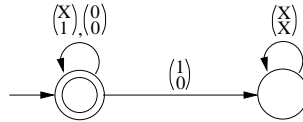
The notation $w[X_i \mapsto M]$ is used for the shortest string that interprets all variables X_j where $j \neq i$ as w does, but interprets X_i as M .

The language $L(\varphi)$ of a formula φ can be defined as the set of satisfying strings: $L(\varphi) = \{w \mid w \models \varphi\}$. By induction in the formula, we can now construct a minimal deterministic finite-state automaton (DFA) A such that $L(A) = L(\varphi)$, where $L(A)$ is the language recognized by A .

For the atomic formulas, we show just one example: the automaton for the formula $\varphi = X_i \text{ sub } X_j$ in the case where $i = 1$ and $j = 2$. The automaton must recognize the language

$$L(X_1 \text{ sub } X_2) = \{w \in (\mathbb{B}^k)^* \mid \text{for all letters in } w: \text{ if the first component is 1, then so is the second} \}$$

Such an automaton is:



The other atomic formulas are treated similarly. The composite formulas are translated as follows:

$\varphi = \sim \varphi'$ Negation of a formula corresponds to automaton complementation. In MONA, this is implemented trivially by flipping accepting and rejecting states.

$\varphi = \varphi' \ \& \ \varphi''$ Conjunction corresponds to language intersection. In MONA, this is implemented with a standard automaton product construction generating only the reachable product states. The resulting automaton is minimized.

$\varphi = \text{ex2 } X_i : \varphi'$ Existential quantification corresponds to a simple quotient operation followed by a projection operation. The quotient operation takes care of the problem that the only strings satisfying φ' may be longer than those satisfying $\text{ex2 } X_i : \varphi'$. The projection operation removes the “track” belonging to X_i , resulting in a nondeterministic automaton, which is subsequently determinized using the subset construction operation, and finally minimized.

This presentation is a simplified version of the procedure actually used in MONA. For more detail, see the MONA User Manual [131].

When the minimal automaton A_0 corresponding to φ_0 has been constructed, validity of φ_0 can be checked simply by observing whether A_0 is the one-state automaton accepting everything. If φ_0 is not valid, a (minimal) counter-example can be constructed by finding a (minimal) path in A_0 from the initial state to a non-accepting state.

WS2S is the generalization of WS1S from linear- to binary-tree-shaped structures [205, 76, 208]. Seen at the “core language” level, WS2S is obtained from WS1S by replacing the single successor predicate by two successor predicates, for *left* and *right* successor respectively. This logic is also decidable by the automaton–logic connection, but using tree automata instead of string automata. The MONA tool also implements this decision procedure.

There is a subtle difference between WS1S, the logic now used in MONA, and M2L(Str), the logic used in early experimental versions [208, 24, 81]. (The difference between WS2S and M2L(Tree) is similar.) In WS1S, formulas are interpreted over *infinite string* models (but quantification is restricted to finite sets only). In M2L(Str), formulas are instead interpreted over *finite string* models. That is, the universe is not the whole set of naturals \mathbb{N} , but a bounded subset $\{0, \dots, n-1\}$, where n is defined by the length of the string. The decision procedure for M2L(Str) is almost the same as for WS1S, only slightly simpler: the quotient operation (before projection) is just omitted. From the language point of view, M2L(Str) corresponds exactly to the regular languages (all formulas correspond to automata *and vice versa*), and WS1S corresponds to those regular languages that are closed under concatenation by 0’s. These properties make M2L(Str) preferable for some applications [18, 194]. However, the fact that not all positions have a successor often makes M2L(Str) rather unnatural to use. Being more closely tied to arithmetic, the WS1S semantics is easier to understand. Also, for instance Presburger Arithmetic can easily be encoded in WS1S whereas there is no obvious encoding in M2L(Str).

Notice that the most significant source of complexity in this decision procedure is the quantifiers, or more precisely, the automaton determinization. Each quantifier can cause an exponential blow-up in the number of automaton states, so in the worst case, this decision procedure has a non-elementary complexity. Furthermore, we cannot hope for a fundamentally better decision procedure since this is also the lower bound for the WS1S decision problem [156]. However, as we will show, even constant factors of improvement can make significant differences in practice.

To make matters even worse (and the challenge the more interesting), the implementation also has to deal with automata with huge alphabets. As mentioned, if φ_0 has k free variables, the alphabet is \mathbb{B}^k . Standard automaton packages cannot handle alphabets of that size, for typical values of k .

6.3 Benchmark Formulas

The experiments presented in the following section are based on twelve benchmark formulas, here shown with their sizes, the logics they are using, and their time and space consumptions when processed by MONA 1.4 (on a 296MHz UltraSPARC with

1GB RAM):

Benchmark	Name	Size	Logic	Time	Space
A	dflipflop.mona	2 KB	WS1S (M2L(Str))	0.4 sec	3 MB
B	euclid.mona	6 KB	WS1S (Presburger)	33.1 sec	217 MB
C	fischer_mutex.mona	43 KB	WS1S	15.1 sec	13 MB
D	html3_grammar.mona	39 KB	WS2S (WSRT)	137.1 sec	208 MB
E	lift_controller.mona	36 KB	WS1S	8.0 sec	15 MB
F	mcnc91_bbsse.mona	9 KB	WS1S	13.2 sec	17 MB
G	reverse_linear.mona	11 KB	WS1S (M2L(Str))	3.2 sec	4 MB
H	search_tree.mona	19 KB	WS2S (WSRT)	30.4 sec	5 MB
I	sliding_window.mona	64 KB	WS1S	40.3 sec	59 MB
J	szymanski_acc.mona	144 KB	WS1S	20.6 sec	9 MB
K	von_neumann_adder.mona	5 KB	WS1S	139.9 sec	116 MB
L	xbar_theory.mona	14 KB	WS2S	136.4 sec	518 MB

The benchmarks have been picked from a large variety of MONA applications ranging from hardware verification to encoding of natural languages.

`dflipflop.mona` – a verification of a D-type flip-flop circuit [18]. Provided by Abdel Ayari.

`euclid.mona` – an encoding in Presburger arithmetic of six steps of reachability on a machine that implements Euclid’s GCD algorithm [198]. Provided by Tom Shiple.

`fischer_mutex.mona` and `lift_controller.mona` – duration calculus encodings of Fischer’s mutual exclusion algorithm and a mine pump controller, translated to MONA code [180]. Provided by Paritosh Pandya.

`html3_grammar.mona` – a tree-logic encoding of the HTML 3.0 grammar annotated with 10 parse-tree formulas [69]. Provided by Niels Damgaard.

`reverse_linear.mona` – verifies correctness of a C program reversing a pointer-linked list [120].

`search_tree.mona` – verifies correctness of a C program deleting a node from a search tree [82].

`sliding_window.mona` – verifies correctness of a sliding window network protocol [200]. Provided by Mark Smith.

`szymanski_acc.mona` – validation of the parameterized Szymanski problem using an accelerated iterative analysis [29]. Provided by Mamoun Filali-Amine.

`von_neumann_adder.mona` and `mcnc91_bbsse.mona` – verification of sequential hardware circuits; the first verifies that an 8-bit von Neumann adder is equivalent to a standard carry-chain adder, the second is a benchmark from MCNC91 [223]. Provided by Sebastian Mödersheim.

`xbar_theory.mona` – encodes a part of a theory of natural languages in the Chomsky tradition. It was used to verify the theory and led to the discovery of mistakes in the original formalization [164]. Provided by Frank Morawietz.

We will use these benchmarks to illustrate the effects of the various implementation “secrets” by comparing the efficiency of MONA shown in the table above with that obtained by handicapping the MONA implementation by not using the techniques.

6.4 Implementation Secrets

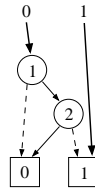
The MONA implementation has been developed and tuned over a period of six years. Many large and small ideas have contributed to a combined speedup of several orders of magnitude. Improvements have taken place at all levels, which we illustrate with the following seven examples from different phases of the compilation and execution of formulas.

To enable comparisons, we summarize the effect of each implementation “secret” by a single dimensionless number for each benchmark formula. Usually, this is simply the speedup factor, but in some cases where the numerator is not available, we argue for a more synthetic measure. If a benchmark cannot run on our machine with 1GB of memory, it is assigned time ∞ .

6.4.1 BDD-based Automata Representation

The very first attempt to implement the decision procedure used a representation based on conjunctive normal forms—however this quickly showed to be very inefficient. The first actually useful version of the MONA tool was the 1995 experimental ML-version [105]. The reason for the success was the novel representation of automata based on (reduced and ordered) BDDs (Binary Decision Diagrams) [47, 48] for addressing the problem of large alphabets. In addition, the representation allows some specialized algorithms to be applied [137, 127].

A BDD is a graph representing a boolean function. The BDD representation has some extremely convenient properties, such as compactness and canonicity, and it allows efficient manipulation. BDDs have successfully been used in a long range of verification techniques (a popular one is [154]). In MONA, a special form of BDDs, called *shared multi-terminal* BDDs, or SMBDDs are used. As an example, the transition function of the tiny automaton shown in Section 6.2 is represented in MONA as the following SMBDD:



The roots and the leaves represent the states. Each root has an edge to the node representing its alphabet part of the transition function. For the other edges, dashed represents 0 and solid represents 1. As an example, from state 0, the transition labeled $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ leads to state 1. In this way, states are still represented explicitly, but the transitions are represented symbolically, in a compact way.

It’s reasonable to ask: “What would happen if we had simply represented the transition tables in a standard fashion, that is, a row for each state and a column for each letter?”. Under this point of view, it makes sense to define a letter for each bit-pattern assignment to the free variables of a subformula (as opposed to the larger set of all variables bound by an outer quantifier). We have instrumented MONA to measure the sum of the number of entries of all such automata transition tables constructed during a run of a version of MONA without BDDs:

	Misses	Table entries	Effect
A	397,472	237,006	0.6
B	48,347,395	2,973,118	0.1
C	46,080,139	1,376,499,745,600	29,871.9
E	19,208,299	290,999,305,488	15,149.7
F	39,942,638	2,844,513,432,416,357,974,016	71,214,961,626,128.9
G	561,202	912,194	1.6
I	95,730,831	116,387,431,997,281,136	1,215,777,934.7
J	24,619,563	15,424,761,908	626.5
K	250,971,828	2,544,758,557,238,438	10,139,618.4

In Section 6.4.2, we describe the importance of cache awareness, which motivates the number of cache misses as a reasonable efficiency measure. “Misses” is the number of cache misses in our BDD-based implementation, and “Table entries” is the total number of table entries in the naive implementation. To roughly estimate the effect of the BDD-representation, we conservatively assume that each table entry results in just a single cache miss; thus, “Effect” compares “Table entries” to “Misses”. The few instances where the effect is less than one correctly identify benchmark formulas where the BDDs are less necessary, but are also artifacts of our conservative assumption. Conversely, the extremely high effects are associated with formulas that could not possibly be decided without BDDs. Of course, the use of BDD-structures completely dominates all other optimizations, since no implementation could realistically be based on the naive table representation.

The BDD-representation was the first breakthrough of the MONA implementation, and the other “secrets” should really be viewed with this as baseline.

6.4.2 Cache-Conscious Data Structures

The data structure used to represent the BDDs for transition functions has been carefully tuned to minimize the number of cache misses that occur. This effort is motivated in earlier work [137], where it is determined that the number of cache misses during unary and binary BDD apply steps totally dominates the running time.

In fact, we argued that if $A1$ is the number of unary apply steps and $A2$ is the number of binary apply steps, then there exist constant m , c_1 , and c_2 such that the total running time is approximately $m(c_1 \cdot A1 + c_2 \cdot A2)$. Here, m is the machine dependent delay incurred by an L2 cache miss, and c_1 and c_2 are the average number of cache misses for unary and binary apply steps. This estimate is based the assumption that time incurred for manipulating auxiliary data structures, such as those used for describing subsets in the determinization construction, is insignificant. For the machine we have used for experiments, it is by a small C utility determined that $m = 0.43\mu s$. In our BDD implementation, explained in [137], we have estimated from algorithmic considerations that $c_1 = 1.7$ and $c_2 = 3$ (the binary apply may entail the use of unary apply steps for doubling tables that were too small—these steps are not counted towards the time for binary apply steps, and that is why we can use the figure $c_2 = 3$); we also estimated that for an earlier conventional implementation, the numbers were $c_1 = 6.7$ and $c_2 = 7.3$. The main reason for this difference is that our specialized package stores nodes directly under their hash address to minimize cache misses; traditional BDD packages store BDD nodes individually with the hash table containing pointers to them—roughly doubling the time it takes to process a node. We no longer support the conventional BDD implementation, so to measure the effect of cache-consciousness,

we must use the above formula to estimate the running times that would have been obtained today.

In the following experiment, we have instrumented MONA to obtain the exact numbers of apply steps:

	Apply1	Apply2	Misses	Auto	Predicted	Conventional	Effect
A	183,949	28,253	397,472	0.2 sec	0.2 sec	0.6 sec	3.0
B	21,908,722	3,700,856	48,347,395	32.8 sec	20.8 sec	74.7 sec	3.6
C	24,585,292	1,428,381	46,080,139	14.2 sec	19.8 sec	75.2 sec	3.8
E	9,847,007	822,796	19,208,299	7.7 sec	8.2 sec	30.9 sec	3.8
F	13,406,047	5,717,453	39,942,638	12.8 sec	17.2 sec	56.6 sec	3.3
G	233,566	54,814	561,504	0.5 sec	0.3 sec	0.8 sec	2.7
I	36,629,195	11,153,733	95,730,831	37.0 sec	41.2 sec	140.5 sec	3.4
J	10,497,759	2,257,791	24,619,563	11.6 sec	10.6 sec	37.3 sec	3.5
K	129,126,447	10,485,623	250,971,828	137.4 sec	107.9 sec	404.7 sec	3.8

“Apply1” is the number of unary apply steps; “Apply2” is the number of binary apply steps; “Misses” is the number of cache misses predicted by the formula above; “Auto” is the part of the actual running time involved in automata constructions; “Predicted” is the running time predicted from the cache misses alone; “Conventional” is the predicted running time for a conventional BDD implementation that was not cache-conscious; and “Effect” is “Conventional” compared to “Predicted”. In most cases, the actual running time is close to the predicted one (within 25%). Note that there are instances where the actual time is about 50% larger than the estimated time: benchmark B involves a lengthy subset construction on an automaton with small BDDs—thus it violates the assumption that the time handling accessory data structures is insignificant; similarly, benchmark G also consists of many automata with few BDD nodes prone to violating the assumption.

In an independent comparison [198] it was noted that MONA was consistently twice as fast as a specially designed automaton package based on a BDD package considered efficient. In [137], the comparison to a traditional BDD package yielded a factor 5 speedup.

6.4.3 Eager Minimization

When MONA inductively translates formulas to automata, a Myhill-Nerode minimization is performed after every product and projection operation. Naturally, it is preferable to operate with as small automata as possible, but our strategy may seem excessive since minimization often exceeds 50% of the total running time. This suspicion is strengthened by the fact that MONA automata by construction contain only reachable states; thus, minimization only collapses redundant states.

Three alternative strategies to the eager one currently used by MONA would be to perform only the very final minimization, only the ones occurring after projection operations, or only the ones occurring after product operations. Many other heuristics could of course also be considered. The following table results from such an investigation:

	Time				Effect
	Only final	After project	After product	Always	
A	∞	∞	0.6 sec	0.4 sec	1.5
B	∞	∞	∞	33.1 sec	∞
C	∞	∞	32.3 sec	15.1 sec	2.1
D	∞	∞	290.6 sec	137.1 sec	2.1
E	∞	∞	19.4 sec	8.0 sec	2.4
F	∞	∞	36.7 sec	13.2 sec	2.8
G	∞	∞	5.8 sec	3.2 sec	1.8
H	∞	∞	59.6 sec	30.4 sec	2.0
I	∞	∞	74.4 sec	40.3 sec	1.8
J	∞	∞	36.3 sec	20.6 sec	1.8
K	∞	∞	142.3 sec	139.9 sec	1.0
L	∞	∞	∞	136.4 sec	∞

“Only final” is the running time when minimization is only performed as the final step of the translation; “After project” is the running time when minimization is also performed after every projection operation; “After product” is the running time when minimization is instead performed after every product operation; “Always” is the time when minimization is performed eagerly; and “Effect” is the “After product” time compared to the “Always” time (since the other two strategies are clearly hopeless). Eager minimization is seen to be always beneficial and in some cases essential for the benchmark formulas.

6.4.4 Guided Tree Automata

Tree automata are inherently computationally more expensive because of their three-dimensional transition tables. We have used a technique of factorization of state spaces to split big tree automata into smaller ones. The basic idea, which may result in exponential savings, is explained in [23, 131]. To exploit this feature, the MONA programmer must manually specify a *guide*, which is a top-down tree automaton that assigns state spaces to the nodes of a tree. However, when using the WSRT logic, a canonical guide is automatically generated. For our two WSRT benchmarks, we measure the effect of this canonical guide:

	Without guide	With guide	Effect
D	584.0 sec	137.1 sec	4.3
H	∞	30.4 sec	∞

“Without guide” shows the running time without any guide, while “With guide” shows the running time with the canonical WSRT guide; “Effect” shows the “Without guide” time compared to the “With guide” time. We have only a small sample space here, but clearly guides are very useful. This is hardly surprising, since they may yield an asymptotic improvement in running time.

6.4.5 DAGification

Internally, MONA is divided into a front-end and a back-end. The front-end parses the input and builds a data structure representing the automata-theoretic operations that will calculate the resulting automaton. The back-end then inductively carries out these operations.

The generated data structure is often seen to contain many common subformulas. This is particularly true when they are compared relative to *signature equivalence*, which holds for two formulas ϕ and ϕ' if there is an order-preserving renaming of the variables in ϕ (increasing with respect to the indices of the variables) such that the representations of ϕ and ϕ' become identical.

A property of the BDD representation is that the automata corresponding to signature-equivalent trees are isomorphic in the sense that only the node indices differ. This means that intermediate results can be reused by simple exchanges of node indices. For this reason, MONA represents the formulas in a DAG (Directed Acyclic Graph), not a tree. The DAG is conceptually constructed from the tree using a bottom-up collapsing process, based on the signature equivalence relation as described in [81].

Clearly, constructing the DAG instead of the tree incurs some overhead, but the following experiments show that the benefits are significantly larger:

	Nodes		Time		Effect
	Tree	DAG	Tree	DAG	
A	2,532	296	1.7 sec	0.4 sec	4.3
B	873	259	79.2 sec	33.1 sec	2.4
C	5,432	461	40.1 sec	15.1 sec	2.7
D	3,038	270	∞	137.1 sec	∞
E	4,560	505	20.5 sec	8.0 sec	2.6
F	1,997	505	49.1 sec	13.2 sec	3.7
G	56,932	1,199	∞	3.2 sec	∞
H	8,180	743	∞	30.4 sec	∞
I	14,058	1,396	107.1 sec	40.3 sec	2.7
J	278,116	6,314	∞	20.6 sec	∞
K	777	273	284.0 sec	139.9 sec	2.0
L	1,504	388	∞	136.4 sec	∞

“Nodes” shows the number of nodes in the representation of the formula. “Tree” is the number of nodes using an explicit tree representation, while “DAG” is the number of nodes after DAGification. “Time” shows the running times for the same two cases. “Effect” shows the “Tree” running time compared to the “DAG” running time. From the differences in the number of nodes, one might expect the total effect to be larger, however DAGification is mainly effective on small formulas where the corresponding automata typically are also smaller. Nevertheless, the DAGification process is seen to provide a substantial and often essential gain in efficiency.

The effects reported sometimes benefit from the fact that the restriction technique presented in the following subsection knowingly generates redundant formulas. This explains some of the failures observed.

6.4.6 Three-Valued Logic and Automata

The standard technique for dealing with first-order variables in monadic second-order logics is to encode them as second-order variables, typically as singletons. However, that raises the issue of *restrictions*: the common phenomenon that a formula ϕ makes sense, relative to some exterior conditions, only when an associated restriction holds. The restriction is also a formula, and the main issue is that ϕ is now essentially undefined outside the restriction. In the case of first-order variables encoded as second-order variables, the restriction is that these variables are singletons. We experienced

the same situation trying to emulate M2L(Str) in WS1S, causing state-space explosions.

The nature of these problems is technical, but fortunately they can be solved through a theory of restriction couched in a three-valued logic [128]. Under this view, a *restricted subformula* φ is associated with a restriction φ_R . If, for some valuation, φ_R does not hold, then formulas containing φ are assigned a new third truth value “don’t-care”. This three-valued logic carries over to the MONA notion of automata—in addition to accept and reject states, they also have “don’t-care” states. A special $\text{restrict}(\varphi_R)$ operation is used for converting reject states to “don’t-care” states for the restriction formulas, and the other automaton operations are modified to ensure that non-acceptance of restrictions is propagated properly.

This gives a cleaner semantics to the restriction phenomenon, and furthermore avoids the state-space explosions mentioned above. According to [128], we can guarantee that the WS1S framework handles all formulas written in M2L(Str), even with intermediate automata that are no bigger than when using the traditional M2L(Str) decision procedure. MONA uses the same technique for the tree logics, WS2S and M2L(Tree).

We refer to [128] for the full theory of three-valued logic and automata. Unfortunately, there is no way of disabling this feature to provide a quantitative comparison.

6.4.7 Formula Reductions

Formula reduction is a means of “optimizing” the formulas in the DAG before translating them into automata. The reductions are based on a syntactic analysis that attempts to identify valid subformulas and equivalences among subformulas.

There are some non-obvious choices here. How should computation resources be apportioned to the reduction phase and to the automata calculation phase? Must reductions guarantee that automata calculations become faster? Should the two phases interact? Our answers are based on some trial and error along with some provisions to cope with subtle interactions with other of our optimization secrets.

MONA 1.4 performs three kinds of formula reductions: 1) simple equality and boolean reductions, 2) special quantifier reductions, and 3) special conjunction reductions. The first kind can be described by simple rewrite rules (only some typical ones are shown):

$$\begin{array}{ll}
 X_i = X_i & \rightsquigarrow \text{true} \\
 \text{true} \& \varphi & \rightsquigarrow \varphi \\
 \text{false} \& \varphi & \rightsquigarrow \text{false} \\
 \varphi \& \varphi & \rightsquigarrow \varphi \\
 \sim\sim\varphi & \rightsquigarrow \varphi \\
 \sim\text{false} & \rightsquigarrow \text{true}
 \end{array}$$

These rewrite steps are guaranteed to reduce complexity, but will not cause significant improvements in running time, since they all either deal with constant size automata or rarely apply in realistic situations. Nevertheless, they are extremely cheap, and they may yield small improvements, in particular on machine generated MONA code.

The second kind of reduction can potentially cause tremendous improvements. As mentioned, the non-elementary complexity of the decision procedure is caused by the automaton projection operations, which stem from quantifiers. The accompanying determinization construction may cause an exponential blow-up in automaton size. Our

basic idea is to apply a rewrite step resembling *let*-reduction, which removes quantifiers:

$$\text{ex2 } X_i : \varphi \rightsquigarrow \varphi[T/X_i] \text{ provided that } \varphi \Rightarrow X_i = T \text{ is valid, and } T \text{ is some term satisfying } FV(T) \subseteq FV(\varphi)$$

where $FV(\cdot)$ denotes the set of free variables. For several reasons, this is not the way to proceed in practice. First of all, finding terms T satisfying the side condition can be an expensive task, in the worst case non-elementary. Secondly, the translation into automata requires the formulas to be “flattened” by introduction of quantifiers such that there are no nested terms. So, if the substitution $\varphi[T/X]$ generates nested terms, then the removed quantifier is recreated by the translation. Thirdly, when the rewrite rule applies in practice, φ usually has a particular structure as reflected in the following more restrictive rewrite rule chosen in MONA:

$$\text{ex2 } X_i : \varphi \rightsquigarrow \varphi[X_j/X_i] \text{ provided that } \varphi \equiv \dots \& X_i = X_j \& \dots \text{ and } X_j \text{ is some variable other than } X_i$$

In contrast to equality and boolean reductions, this rule is not guaranteed to improve performance, since substitutions may cause the DAG reuse degree to decrease.

The third kind of reduction applies to conjunctions, of which there are two special sources. One is the formula flattening just mentioned; the other is the formula restriction technique mentioned in Section 6.4.6. Both typically introduce many new conjunctions. Studies of a graphical representation of the formula DAGs (MONA can create such graphs automatically) led us to believe that many of these new conjunctions are redundant. A typical rewrite rule addressing such redundant conjunctions is the following:

$$\varphi_1 \& \varphi_2 \rightsquigarrow \varphi_1 \text{ provided that } \text{nonrestr}(\varphi_2) \subseteq \text{nonrestr}(\varphi_1) \cup \text{restr}(\varphi_1) \text{ and } \text{restr}(\varphi_2) \subseteq \text{restr}(\varphi_1)$$

Here, $\text{restr}(\varphi)$ is the set of $\text{restrict}(\cdot)$ conjuncts in φ , and $\text{nonrestr}(\varphi)$ is the set of $\text{non-restrict}(\cdot)$ conjuncts in φ . This reduction states that it is sufficient to assert φ_1 when $\varphi_1 \& \varphi_2$ was originally asserted in situations where the non-restricted conjuncts of φ_2 are already conjuncts of φ_1 —whether restricted or not—and the restricted conjuncts of φ_2 are also restricted conjuncts of φ_1 .

All rewrite rules mentioned here have the property that they cannot “do any harm”, that is, have a negative impact on the automaton sizes. (They can however damage the reuse factor obtained by the DAGification, but this is rarely a problem in practice.) A different kind of rewrite rules could be obtained using heuristics—this will be investigated in the future.

With the DAG representation of formulas, the reductions just described can be implemented relatively easily in MONA. The table below shows the effects of performing the reductions on the benchmark formulas:

	Hits			Time					Effect
	Simple	Quant.	Conj.	None	Simple	Quant.	Conj.	All	
A	12	8	22	0.8 sec	0.7 sec	0.7 sec	0.7 sec	0.4 sec	2.0
B	10	45	0	58.2 sec	58.8 sec	56.2 sec	56.8 sec	33.1 sec	1.8
C	9	13	8	43.7 sec	41.9 sec	37.1 sec	42.9 sec	15.1 sec	2.9
D	4	28	27	542.7 sec	536.1 sec	296.0 sec	404.7 sec	137.1 sec	4.0
E	5	6	19	22.6 sec	23.4 sec	16.6 sec	22.7 sec	8.0 sec	2.8
F	3	1	1	28.3 sec	29.9 sec	27.0 sec	27.2 sec	13.2 sec	2.1
G	65	318	191	6.1 sec	5.9 sec	6.1 sec	5.9 sec	3.2 sec	1.9
H	35	32	81	104.1 sec	102.6 sec	71.0 sec	98.5 sec	30.4 sec	3.4
I	102	218	7	76.2 sec	76.5 sec	75.0 sec	76.0 sec	40.3 sec	1.9
J	91	0	1	37.3 sec	37.9 sec	37.6 sec	37.0 sec	20.6 sec	1.9
K	9	4	1	313.7 sec	267.9 sec	240.3 sec	302.6 sec	139.9 sec	2.3
L	4	4	18	∞	∞	∞	∞	136.4 sec	∞

“Hits” shows the number of times each of the three kinds of reduction is performed; “Time” shows the total running time in the cases where no reductions are performed, only the first kind of reductions are performed, only the second, only the third, and all of them together. “Effect” shows the “None” times compared to the “All” times. All benchmarks gain from formula reductions, and in a single example this technique is even necessary. Note that most often all three kinds of reductions must act in unison to obtain significant effects.

A general benefit from formula reductions is that tools generating MONA formulas from other formalisms may generate naive and voluminous output while leaving optimizations to MONA. In particular, tools may use existential quantifiers to bind terms to fresh variables, knowing that MONA will take care of the required optimization.

6.5 Future Developments

Several of the techniques described in the previous section can be further refined of course. The most promising ideas seem however to concentrate on the BDD representation. In the following, we describe three such ideas.

It is a well-known fact [47] that the ordering of variables in the BDD automata representation has a strong influence on the number of BDD nodes required. The impact of choosing a good ordering can be an exponential improvement in running times. Finding the optimal ordering is an NP-complete problem, but we plan to experiment with the heuristics that have been suggested [53].

We have sometimes been asked: “Why don’t you encode the states of the automata in BDDs, since that is a central technique in model checking?”. The reason is that there is no obvious structure to the state space in most cases that would lend itself towards an efficient BDD representation. For example, consider the consequences of a subset construction or a minimization construction, where similar states are collapsed; in either case, it is not obvious how to represent the new state. However, the ideas are worth investigating.

For our tree automata, we have experimentally observed that the use of guides produce a large number of component automata many of which are almost identical. We will study how to compress this representation using a BDD-like global structure.

6.6 Conclusion

The presented techniques reflect a lengthy Darwinian development process of the MONA tool in which only robust and useful ideas have survived. We have not mentioned here the many ideas that failed or were surpassed by other techniques. Our experiences confirm the maxim that optimizations must be carried out at all levels and that no single silver bullet is sufficient. We are confident that further improvements are still possible and that other interesting applications will be made.

Acknowledgments

Many people have contributed to the development of MONA, in particular we are grateful to David Basin, Morten Biehl, Jacob Elgaard, Jesper Gulmann, Jacob Jensen, Michael Jørgensen, Bob Paige, Theis Rauhe, and Anders Sandholm. We also thank the MONA users who kindly provided the benchmark formulas.

Chapter 7

Compile-Time Debugging of C Programs Working on Trees

with Jacob Elgaard and Michael I. Schwartzbach

Abstract

We exhibit a technique for automatically verifying the safety of simple C programs working on tree-shaped data structures. We do not consider the complete behavior of programs, but only attempt to verify that they respect the shape and integrity of the store. A verified program is guaranteed to preserve the tree-shapes of data structures, to avoid pointer errors such as NULL dereferences, leaking memory, and dangling references, and furthermore to satisfy assertions specified in a specialized store logic.

A program is transformed into a single formula in WSRT, an extension of WS2S that is decided by the MONA tool. This technique is complete for loop-free code, but for loops and recursive functions we rely on Hoare-style invariants. A default well-formedness invariant is supplied and can be strengthened as needed by programmer annotations. If a program fails to verify, a counterexample in the form of an initial store that leads to an error is automatically generated.

This extends previous work that uses a similar technique to verify a simpler syntax manipulating only list structures. In that case, programs are translated into WS1S formulas. A naive generalization to recursive data-types determines an encoding in WS2S that leads to infeasible computations. To obtain a working tool, we have extended MONA to directly support recursive structures using an encoding that provides a necessary state-space factorization. This extension of MONA defines the new WSRT logic together with its decision procedure.

7.1 Introduction

Catching pointer errors in programs is a difficult task that has inspired many assisting tools. Traditionally, these come in three flavors. First, tools such as Purify [188] and Insure++ [142] instrument the generated code to monitor the runtime behavior thus indicating errors and their sources. Second, traditional compiler technologies such as program slicing [210], pointer analysis [96], and shape analysis [192] are used in

tools like CodeSurfer [99] and Aspect [117] that conservatively detect known causes of errors. Third, full-scale program verification is attempted by tools like LCLint [84] and ESC [74], which capture runtime behavior as formulas and then appeal to general theorem provers.

All three approaches lead to tools that are either incomplete or unsound (or both), even for straight-line code. In practice, this may be perfectly acceptable if a significant number of real errors are caught.

In previous work [120], we suggest a different balance point by using a less expressive program logic for which Hoare triples on loop-free code is decidable when integer arithmetic is ignored. That work is restricted by allowing only a `while`-language working on linear lists. In the present paper we extend our approach by allowing recursive functions working on recursive data-types. This generalization is conceptually simple but technically challenging, since programs must now be encoded in WS2S rather than the simpler WS1S. Decision procedures for both logics are provided by the MONA tool [131, 158] on which we rely, but a naive generalization of the previous encoding leads to infeasible computations. We have responded by extending MONA to directly support a logic of recursive data-types, which we call WSRT. This logic is encoded in WS2S in a manner that exploits the internal representation of MONA automata to obtain a much needed state-space factorization.

Our resulting tool catches all pointer errors, including NULL dereferences, leaking memory, and dangling references. It can also verify assertions provided by the programmer in a special store logic. The tool is sound and complete for loop-free code including `if`-statements with restricted conditions: it will reject exactly the code that may cause errors or violate assertions when executed in some initial store. For `while`-loops or functions, the tool relies on annotations in the form of invariants and pre- and post-conditions. In this general case, our tool is sound but incomplete: safe programs exist that cannot be verified regardless of the annotations provided. In practical terms, we provide default annotations that in many cases enable verification.

Our implementation is reasonably efficient, but can only handle programs of moderate sizes, such as individual operations of data-types. If a program fails to verify, a counterexample is provided in the form of an initial store leading to an error. A special simulator is supplied that can trace the execution of a program and provide graphical snapshots of the store. Thus, a reasonable form of compile-time debugging is made available. While we do not detect all program errors, the verification provided serves as a finely masked filter for most bugs.

As an example, consider the following recursive data-type of binary trees with red, green, or blue nodes:

```
struct RGB {
    enum {red,green,blue} color;
    struct RGB *left;
    struct RGB *right;
};
```

The following non-trivial application collects all green leaves into a right-linear tree and changes all the blue nodes to become red:

```
/**data**/ struct RGB *tree;
```

```

/**data*/ struct RGB *greens;

enum bool {false,true};

enum bool greenleaf(struct RGB *t) {
  if (t==0) return false;
  if (t->color!=green) return false;
  if (t->left!=0 || t->right!=0) return false;
  return true;
}

void traverse(struct RGB *t) {
  struct RGB *x;
  if (t!=0) {
    if (t->color==blue) t->color = red;
    if (greenleaf(t->left)==true /**keep: t!=0 **/) {
      t->left->right = greens;
      greens = t->left;
      t->left=0;
    }
    if (greenleaf(t->right)==true /**keep: t!=0 **/) {
      t->right->right = greens;
      greens = t->right;
      t->right=0;
    }
    traverse(t->left); /**keep: t!=0 **/
    traverse(t->right); /**keep: t!=0 **/
  }
}

/**pre: greens==0 **/
main() { traverse(tree); }

```

The special comments are assertions that the programmer must insert to specify the intended model (*/**data**/*), restrict the set of stores under consideration (*/**pre**/*), or aid the verifier (*/**keep**/*). They are explained further in Section 7.2.4.

Without additional annotations, our tool can verify this program (in 33 seconds on a 266MHz Pentium II PC with 128 MB RAM). This means that no pointer errors occur during execution from any initial store. Furthermore, both *tree* and *greens* are known to remain well-formed trees. Using the assertion:

```
all p: greens(->left + ->right)*==p => (p!=0 => p->color==green)
```

we can verify (in 74 seconds) that *greens* after execution contains only green nodes. That *greens* is right-linear is expressed through the assertion:

```
all p: greens(->left + ->right)*==p => (p!=0 => p->left==0)
```

In contrast, if we assert that *greens* ends up empty, the tool responds with a minimal counterexample in the form of an initial store in which *tree* contains a green leaf.

An example of the simulator used in conjunction with counterexamples comes from the following fragment of an implementation of red-black search trees. Consider

the following program, which performs a left rotation of a node n with parent p in such a tree:

```
struct Node {
    enum {red, black} color;
    struct Node *left;
    struct Node *right;
};

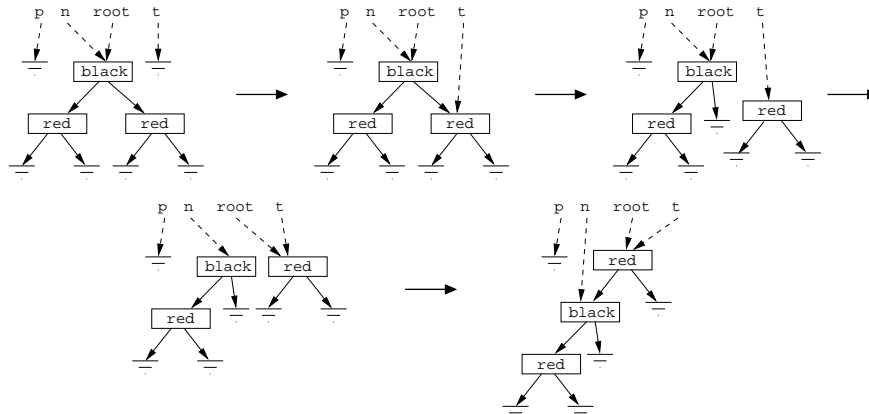
/**data**/ struct Node *root;

/**pre: n!=0 & n->right!=0 &
    (p!=0 => (p->left==n | p->right==n)) &
    (p==0 => n==root) **/
void left_rotate(struct Node *n, struct Node *p) {
    struct Node *t;
    t = n->right;
    n->right = t->left;
    if (n==root) root = t;
    else if (p->left==n) p->left = t;
    else p->right = t;
    t->left = n;
}
```

In our assertion language, we cannot express the part of the red-black data-type invariant that each path from the root to a leaf must contain the same number of black nodes; however we can capture the part that the root is black and that a red node cannot have red children:

```
root->color==black &
all p: p->color==red =>
    (p->left->color!=red & p->right->color!=red)
```

If we add the above assertion as a data-type invariant, we are (in 18 seconds) given a counterexample. If we apply the simulator, we see the following example run, which shows that we have forgotten to consider that the root may become red (in which case we should add a line of code coloring it black):



Such detailed feedback at compile-time is clearly a useful debugging tool.

7.2 The Language

The language in consideration is a simple yet non-trivial subset of C. It allows declaration of tree-shaped recursively typed data structures and recursive imperative functions operating on the trees. The subset is chosen such that the verification we intend to perform becomes decidable. Thus, for instance, integer arithmetic is omitted from the language; only finite enumeration types can be expressed. Also, to smoothen presentation, many other C constructs have been omitted although some of them easily could be included directly, and other by approximating their behavior.

We begin by defining the core language. After that, we describe how programs can be annotated with formulas expressing additional requirements for correctness.

7.2.1 The C Subset

The abstract syntax of the C subset is defined using EBNF notation, where furthermore \otimes is used to denote comma-separated lists with zero or more elements. The semantics of the language is as known from C.

A program consists of declarations of structures, enumerations, variables, and functions:

$$\text{program} \rightarrow (\text{struct} \mid \text{enum} \mid \text{var} \mid \text{function})^*$$

A structure contains an enumeration denoting its value and a union of structures containing pointers to its child structures. An enumeration is a list of identifiers:

$$\begin{aligned} \text{struct} &\rightarrow \text{struct } id \{ \\ &\quad \text{enum } id \text{ } id; \\ &\quad \text{union } \{ \\ &\quad \quad (\text{struct } \{ \\ &\quad \quad \quad (\text{struct } id * id;)^* \\ &\quad \quad \quad \} id;)^* \\ &\quad \} id; \\ &\quad \}; \\ \text{enum} &\rightarrow \text{enum } id \{ id^+ \}; \end{aligned}$$

The enumeration values denote the *kind* of the structure, and the kind determines which is the *active* union member. The association between enumeration values and union members is based on their indices in the two lists. Such data structures are typical in real-world C programs and exactly define recursive data-types. One goal of our verification is to ensure that only active union members are accessed.

For abbreviation we allow declarations of structures and enumerations to be inlined. Also, we allow $(\text{struct } id * id;)^*$ in place of $\text{union } \{ \dots \}$, implicitly meaning that all union members are identical. A variable is either a pointer to a structure or an enumeration:

$$\begin{aligned} \text{var} &\rightarrow \text{type } id; \\ \text{type} &\rightarrow \text{struct } id * \mid \text{enum } id \end{aligned}$$

A function can contain variable declarations and statements:

$$\begin{aligned}
 \text{function} \quad \rightarrow \quad & (\text{void} \mid \text{type}) \text{id} ((\text{type id})^{\circledast}) \{ \\
 & \quad \text{var}^* \text{stm}^? \\
 & \quad (\text{return rvalue};)^? \\
 & \}
 \end{aligned}$$

A statement is a sequence, an assignment, a function call, a conditional statement, a while-loop, or a memory deallocation:

$$\begin{aligned}
 \text{stm} \quad \rightarrow \quad & \text{stm stm} \mid \\
 & \text{lvalue} = \text{rvalue}; \mid \\
 & \text{id} ((\text{rvalue})^{\circledast}); \mid \\
 & \text{if} (\text{cond}) \text{stm} (\text{else stm})^? \mid \\
 & \text{while} (\text{cond}) \text{stm} \mid \\
 & \text{free} (\text{lvalue});
 \end{aligned}$$

A condition is a boolean expression evaluating to either true or false; the expression ? represents non-deterministic choice and can be used in place of those C expressions that are omitted from our subset language:

$$\text{cond} \quad \rightarrow \quad \text{cond} \& \text{cond} \mid \text{cond} \mid \text{cond} \mid ! \text{cond} \mid \text{rvalue} == \text{rvalue} \mid ?$$

An *lvalue* is an expression designating an enumeration variable or a pointer variable. An *rvalue* is an expression evaluating to an enumeration value or to a pointer to a structure. The constant 0 is the NULL pointer, `malloc` allocates memory on the heap, and `id(...)` is a function call:

$$\begin{aligned}
 \text{lvalue} \quad \rightarrow \quad & \text{id} (\rightarrow \text{id} (. \text{id})^?)^* \\
 \text{rvalue} \quad \rightarrow \quad & \text{lvalue} \mid 0 \mid \text{malloc}(\text{sizeof}(\text{id})) \mid \text{id}(\text{rvalue}^{\circledast})
 \end{aligned}$$

The nonterminal *id* represents identifiers.

The presentation of our verification technique is based on C for familiarity reasons only—no intrinsic C constructs are utilized.

7.2.2 Modeling the Store

During execution of a program, structures located in the heap are allocated and freed, and field variables and local variables are assigned values. The state of an execution can be described by a model of the heap and the local variables, called the *store*.

A store is modeled as a finite graph, consisting of a set of *cells* representing structures, a distinguished *NULL cell*, a set of *program variables*, and *pointers* from cells or program variables to cells. Each cell is labeled with a *value* taken from the enumerations occurring in the program. Furthermore, each cell can have a *free* mark, meaning that it is currently not allocated.

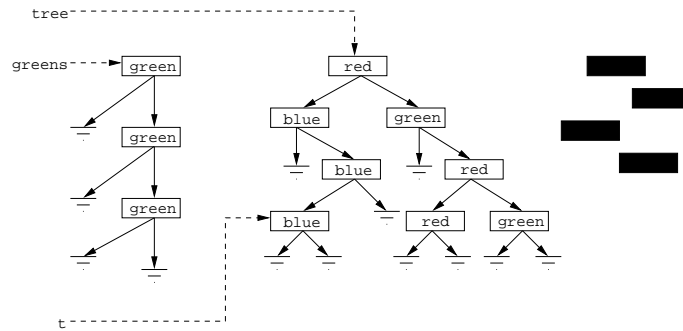
Program variables are those that are declared in the program either globally or inside functions. To enable the verification, we need to classify these variables as either *data* or *pointer* variables. A variable is classified as a data variable by prefixing its declaration in the program with the special comment `/**data**/`; otherwise, it is considered a pointer variable.

A store is *well-formed* if it satisfies the following properties:

- the cells and pointers form disjoint tree structures (the NULL cell may be shared, though);
- each data variable points either to the root of a tree or to the NULL cell;
- each pointer variable points to any cell (including the NULL cell);
- a cell is marked as free if and only if it is not reachable from a program variable; and
- the type declarations are respected—this includes the requirement that a cell representing a structure has an outgoing pointer for each structure pointer declared in its active union member.

With the techniques described in the remainder of this paper, it is possible to automatically verify whether well-formedness is preserved by all functions in a given program. Furthermore, additional user defined properties expressed in the logic presented in Section 7.2.3 can be verified.

The following illustrates an example of a well-formed store containing some RGB-trees as described in Section 7.1. Tree edges are solid lines whereas the values of pointer variables are dashed lines; free cells are solid black:



7.2.3 Store Logic

Properties of stores can conveniently be stated using logic. The declarative and succinct nature of logic often allows simple specifications of complex requirements. The logic presented here is essentially a first-order logic on finite tree structures [208]. It has the important characteristic of being decidable, which we will exploit for the program verification.

A formula ϕ in the store logic is built from boolean connectives, first-order quantifiers, and basic propositions. A term t denotes either an enumeration value or a pointer to a cell in the store. A path set P represents a set of paths, where a path is a sequence of pointer dereferences and union member selections ending in either a pointer or an enumeration field. The signature of the logic consists of dereference functions, path relations, and the relations `free` and `root`:

$$\begin{aligned}
 \phi \quad \rightarrow \quad & ! \phi \mid \phi \& \phi \mid \phi \mid \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \\
 & \text{ex id} : \phi \mid \text{all id} : \phi \mid \text{true} \mid \text{false} \mid \\
 & \text{id}(P)^? == t \mid \text{free}(t) \mid \text{root}(t)
 \end{aligned}$$

A path relation, $id \ P == \ t$, compares either enumeration values or cell pointers. The identifier id may be either a bound quantified variable, a program variable, or an enumeration value, and t is a term.

If both id and t denote cell pointers, a path relation is true for a given store if there is a path in P from the cell denoted by id to the cell denoted by t in the store. If P is omitted, the relation is true if id and t denote the same cell.

If id denotes a cell pointer and t is an enumeration value, a path relation is true for a given store if there is a path satisfying P from the cell denoted by id to an enumeration field with the value t in the store.

The relation $free(t)$ is true in a given store if the cell denoted by t is marked as not allocated in the store. The relation $root(t)$ is true if t denotes the root of some tree.

A term is a sequence of applications of the dereference function and union member selections or the constant 0 representing the special NULL cell:

$$t \rightarrow id (\rightarrow id (. id)^?)^* \mid 0$$

A path set is a regular expression:

$$P \rightarrow \rightarrow id (. id)^? \mid P + P \mid P P \mid P *$$

The path set defined by $\rightarrow id_1 . id_2$ consists of a single dereference of id_1 and subsequent selection of the member id_2 . The expressions $P + P$, $P P$, and $P *$ respectively denote union, concatenation, and Kleene star.

7.2.4 Program Annotations and Hoare Triples

The verification technique is based on Hoare triples [106], that is, constructs of the form $\{\varphi_1\}stm\{\varphi_2\}$. The meaning of this triple is that executing the statement stm in a store satisfying the pre-condition φ_1 always results in a store satisfying the post-condition φ_2 , provided that the statement terminates. Well-formedness is always implicitly included in both φ_1 and φ_2 . We can only directly decide such triples for loop-free code. Programs containing loops—either as `while`-loops or as function calls—must be split into loop-free fragments.

A program can be annotated with formulas expressing requirements for correctness using a family of designated comments. These annotations are also used to split the program into a set of Hoare triples that subsequently can be verified separately.

`/**pre: φ */` and `/**post: φ */` may be placed between the signature and the body of a function. The `pre` formula expresses a property that the verifier may assume initially holds when the function is executed. The `post` formula expresses a property intended to hold after execution of the function. The states before and after execution may be related using otherwise unused variables.

`/**inv: φ */` may be placed between the condition and the body of a `while`-loop. It expresses an invariant property that must hold before execution of the loop and after each iteration. It splits the code into three parts: the statements preceding the `while`-loop, its body, and the statements following it.

`/**keep: ϕ */` may be placed immediately after a function call. It expresses a property that must hold both before and after the call. It splits the code into two parts: the statements before and after the call. The `keep` formulas can specify invariants for recursive function calls just as `inv` formulas can specify invariants for `while`-loops.

`/**assert: ϕ */` may be placed between statements. It splits the statement sequence, such that a Hoare triple is divided into two smaller triples, where the post-condition of the first and the pre-condition of the second both are ϕ . This allows modular analysis to be performed. The variant `/**assert: ϕ assume: ϕ */` allows the post-condition and the pre-condition to be different, and thereby to weaken the verification requirements. This is needed whenever a sufficiently strong property either cannot be expressed or requires infeasible computations.

`/**check: ϕ */` *stm* informally corresponds to “if ($\neg\phi$) fail; else *stm*”, where *fail* is some statement that fails to verify. This can be used to check that a certain property holds without creating two Hoare triples incurring a potential loss of information.

Whenever a pre- or post-condition, an invariant, or a `keep`-formula is omitted, the default formula `true` is implicitly inserted. Actually, many interesting properties can be verified with just these defaults. As an example, the program:

```
/**data*/ struct RGB *x;
struct RGB *p;
struct RGB *q;

p = x;
q = 0;
while (p!=0 & q==0) /**inv: q!=0 => q->color==red */ {
    if (p->color==red) q = p;
    else if (p->color==green) p = p->left;
    else /**assert: p->color==blue */ p = p->right;
}
```

yields the following set of Hoare triples and logical implications to be checked:

```
{ true } p = x; q = 0; { I }
(I & !B) => true
{ I & B & B1 } q = p; { I }
{ I & B & !B1 & B2 } p = p->left; { I }
(I & B & !B1 & !B2) => (p->color==blue)
{ I & B & !B1 & !B2 & p->color==blue } p = p->right; { I }
```

where *B* is the condition of the `while`-loop, *I* is the invariant, *B1* is the condition of the outer `if`-statement and *B2* that of the inner `if`-statement. Note that the generated Hoare triples are completely independent of each other—when a triple is divided into two smaller triples, no information obtained from analyzing the first triple is used when analyzing the second.

7.3 Deciding Hoare Triples

The generated Hoare triples and logical implications—both the formula parts and the program parts—can be encoded in the logic WS2S which is known to be decidable. This encoding method follows directly from [120] by generalizing from list structures to tree structures in the style of [140]. The MONA tool provides an implementation of a decision procedure for WS2S, so in principle making a decision procedure for the present language requires no new ideas.

As we show in the following, this method will however lead to infeasible computations making it useless in practice. The solution is to exploit the full power of the MONA tool: usually, WS2S is decided using a correspondence with ordinary tree automata—MONA uses a representation called *Guided Tree Automata*, which when used properly can be exponentially more efficient than ordinary tree automata. However, such a gain requires a non-obvious encoding.

We will not describe how plain MONA code directly can be generated from the Hoare triples and logical implications. Instead we introduce a logic called *WSRT*, *weak monadic second-order logic with recursive types*, which separates the encoding into two parts: the Hoare triples and logical implications are first encoded in WSRT, and then WSRT is translated into basic MONA code. This has two benefits: WSRT provides a higher level of abstraction for the encoding task, and, as a by-product, we get an efficient implementation of a general tree logic which can be applied in many other situations where WS2S and ordinary tree automata have so far been used.

7.3.1 Weak Monadic Second-Order Logic with Recursive Types

A *recursive type* is a set of recursive equations of the form:

$$T_i = v_1(c_{1,1} : T_{j_{1,1}}, \dots, c_{1,m_1} : T_{j_{1,m_1}}), \dots, v_n(c_{n,1} : T_{j_{n,1}}, \dots, c_{n,m_n} : T_{j_{n,m_n}})$$

Each T denotes the name of a type, each v is called a *variant*, and each c is called a *component*. A tree conforms to a recursive type T if its root is labeled with a variant v from T and it has a successor for each component in v such that the successor conforms to the type of that component. Note that types defined by `structs` in the language in Section 7.2.1 exactly correspond to such recursive types.

The logic WSRT is a weak monadic second-order logic. Formulas are interpreted relative to a set of trees conforming to recursive types. Each node is labeled with a variant from a recursive type. A tree variable denotes a tree conforming to a fixed recursive type. A first-order variable denotes a single node. A second-order variable denotes a finite set of nodes.

A formula is built from the usual boolean connectives, first-order and weak monadic second-order quantifiers, and the special WSRT basic formulas:

$type(t, T)$ which is true iff the first-order term t denotes a node which is labeled with some variant from the type T ; and

$variant(t, x, T, v)$ which is true iff the tree denoted by the tree variable x at the position denoted by t is labeled with the T variant v .

Second-order terms are built from second-order variables and the set operations union, intersection and difference. First-order terms are built from first-order variables and the special WSRT functions:

$tree_root(x)$ which evaluates to the root of the tree denoted by x ; and

$succ(t, T, v, c)$ which, provided that the first-order term t denotes a node of the T variant v , evaluates to its c component.

This logic corresponds to the core of the FIDO language [140] and is also reminiscent of the LISA language [12]. It can be reduced to WS2S and thus provides no more expressive power, but we will show that a significantly more efficient decision procedure exists if we bypass WS2S.

7.3.2 Encoding Stores and Formulas in WSRT

The idea behind the decision procedure for Hoare triples is to encode well-formed stores as trees. The effect of executing a loop-free program fragment is then in a finite number of steps to transform one tree into another. WSRT can conveniently be used to express regular sets of finite trees conforming to recursive types, which turns out to be exactly what we need to encode pre- and post-conditions and effects of execution.

We begin by making some observations that simplify the encoding task. First, note that NULL pointers can be represented by adding a “NULL kind” with no successors to all structures. Second, note that memory allocation issues can be represented by having a “free list” for each struct, just as in [120]. We can now represent a well-formed store by a set of WSRT variables:

- each data variable is represented by a WSRT tree variable with the same recursive type, where we use the fact that the types defined by structs exactly correspond to the WSRT notion of recursive types; and
- each pointer variable in the program is represented by a WSRT first-order variable.

For each program point, a set of WSRT predicates called *store predicates* is used to express the possible stores:

- for each data variable d in the program, the predicate $root_d(t)$ is true whenever the first-order term t denotes the root of d ;
- for each pointer variable p , the predicate $pos_p(t)$ is true whenever t and p denote the same position;
- for each pointer field f occurring in a union u in some structure s , the predicate $succ_{f,u,s}(t_1, t_2)$ is true whenever the first-order term t_1 points to a cell of type s having the value u , and the f component of this cell points to the same node as the first-order term t_2 ;
- for each possible enumeration value e , the predicate $kind_e(t)$ is true whenever t denotes a cell with value e ; and

- to encode allocation status, the predicate $free_s(t)$ is true whenever t denotes a non-allocated cell.

A set of store predicates called the *initial store predicates* defining a mapping of the heap into WSRT trees can easily be expressed in the WSRT logic. For instance, the initial store predicates *root*, *succ*, and *kind* simply coincide with the corresponding basic WSRT constructs.

Based on a set of store predicates, the well-formedness property and all store-logic formulas can be encoded as other predicates. For well-formedness, the requirements of the recursive types are expressed using the *root*, *kind*, and *succ* predicates, and the requirement that all data structures are disjoint trees is a simple reachability property. For store-logic formulas, the construction is inductive: boolean connectives and quantifiers are directly translated into WSRT; terms are expressed using the store predicates *root*, *kind*, and *succ*; and the basic formulas $free(t)$ and $root(t)$ can be expressed using the store predicates *free* and *root*. Only the regular path sets are non-trivial; they are expressed in WSRT using the method from [138] (where path sets are called “routing expressions”). Note that even though the logic in Section 7.2.3 is a first-order logic, we also need the weak monadic second-order fragment of WSRT to express well-formedness and path sets.

7.3.3 Predicate Transformation

When the program has been broken into loop-free fragments, the Hoare triples are decided using the transduction technique introduced in [139]. In this technique, the effect of executing a loop-free program fragment is simulated, step by step, by transforming store predicates accordingly, as described in the following.

Since the pre-condition of a Hoare triple always implicitly includes the well-formedness criteria, we encode the set of *pre-stores* as the conjunction of well-formedness and the pre-condition, both encoded using the initial store predicates, and we initiate the transduction with the initial store predicates. For each step, a new set of store predicates is defined representing the possible stores after executing that step. This predicate transformation is performed using the same ideas as in [120], so we omit the details.

When all steps in this way have been simulated, we have a set of *final* store predicates which exactly represents the changes made by the program fragment. We now encode the set of *post-stores* as the conjunction of well-formedness and the post-condition, both encoded using the final store predicates. It can be shown that the resulting predicate representing the post-stores coincides with the weakest precondition of the code and the post-condition. The Hoare triple is satisfied if and only if the encoding of the pre-stores implies the encoding of the post-stores.

Our technique is sound: if verification succeeds, the program is guaranteed to contain no errors. For loop-free Hoare triples, it is also complete. That is, every effect on the store can be expressed in the store logic, and this logic is decidable. In general, no approximation takes place—all effects of execution are simulated precisely. Nevertheless, since not all true properties of a program containing loops can be expressed in the logic, the technique is, in general, not complete for whole programs.

7.4 Deciding WSRT

As mentioned, there is a simple reduction from WSRT to WS2S, and WS2S can be decided using a well-known correspondence between WS2S formulas and ordinary tree automata. The resulting so-called *naive* decision procedure for WSRT is essentially the same as the ones used in FIDO and LISA and as the “conventional encoding of parse trees” in [69]. The naive decision procedure along with its deficiencies is described in Section 7.4.1. In Section 7.4.2 we show an efficient decision procedure based on the more sophisticated notion of guided tree automata.

7.4.1 The Naive Decision Procedure

WS2S, the weak monadic second-order theory of two successors, is a logic that is interpreted relative to a binary tree. A first-order variable denotes a single node in the tree, and a second-order variable denotes a finite set of nodes. For a full definition of WS2S, see [208] or [131].

The decision procedure implemented in MONA inductively constructs a tree automaton for each sub-formula, such that the set of trees accepted by the automaton is the set of interpretations that satisfy the sub-formula. This decision procedure not only determines validity of formulas; it also allows construction of counterexamples whenever a formula is not valid.

Note that the logic WS_nS , where each node has n successors instead of just two, easily can be encoded in WS2S by replacing each node with a small tree with n leaves. The idea in the encoding is to have a one-to-one mapping from nodes in a WSRT tree to nodes in a WS_nS tree, where we choose n as the maximal fanout of all recursive types.

Each WSRT tree variable x is now represented by b second-order variables v_1, \dots, v_b where b is the number of bits needed to encode the possible type variants. For each node in the n -ary tree, membership in $v_1 \dots v_b$ represents some binary encoding of the label of the corresponding node in the x tree.

Using this representation, all the basic WSRT formulas and functions can now easily be expressed in WS_nS . We omit the details. For practical applications, this method leads to intractable computations requiring prohibitive amounts of time and space. Even a basic concept such as *type well-formedness* yields immense automata. Type well-formedness is the property that the values of a given set of WS2S variables do represent a tree of a particular recursive type.

This problem can be explained as follows. The WS2S encoding is essentially the same as the “conventional encoding of parse trees” in [69], and type well-formedness corresponds to grammar well-formedness. In that paper, it is shown that the number of states in the automaton corresponding to the grammar well-formedness predicate is linear in the size of the grammar, which in our case corresponds to the recursive types. As argued e.g. in [126], tree automata are at least quadratically more difficult to work with than string automata, since the transition tables are two-dimensional as opposed to one-dimensional. This inevitably causes a blowup in time and space requirements for the whole decision procedure.

By this argument, it would be pointless making an implementation based on the described encoding. This claim is supported by experiments with some very simple

examples; in each case, we experienced prohibitive time and space requirements.

7.4.2 A Decision Procedure using Guided Tree Automata

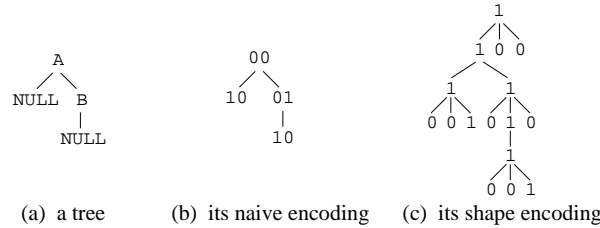
The MONA implementation of WS2S provides an opportunity to factorize the state-space and hence make implementation feasible. To exploit this we must, however, change the encoding of WSRT trees, as described in the following.

The notion of guided tree automata (GTA) was introduced in [23] to combat state-space explosions and is now fully implemented in MONA [131]. A GTA is a tree automaton equipped with separate state spaces that—independently of the labeling of the tree—are assigned to the tree nodes by a top-down automaton, called the *guide*. The secret behind a good factorization is to create the right guide.

A recursive type is essentially also a top-down automaton, so the idea is to derive a guide from the recursive types. This is however not possible with the naive encoding, since the type of a WS n S node depends on the actual value of its parent node.

Instead of using the one-to-one mapping from WSRT tree nodes to WS n S tree nodes labeled with type variants, we represent a WSRT tree entirely by the *shape* of a WS n S tree, similarly to the “shape encoding” in [69]. Each node in the WSRT tree is represented by a WS n S node with a successor node for each variant, and each of these nodes have themselves a successor for each component in the variant. A WSRT tree is then represented by a *single* second-order WS n S variable whose value indicates the active variants.

The following illustrates an example of a tree conforming to the recursive type $\text{Tree} = A(\text{left}:\text{Tree}, \text{right}:\text{Tree}), B(\text{next}:\text{Tree}), \text{NULL}$ and its encodings:



This encoding has the desired property that a WS n S tree position always is assigned the same type, independently of the tree values, so a GTA guide can directly be derived from the types. This guide factorizes the state space such that all variants and components in the recursive types have their own separate state spaces. Furthermore, the intermediate nodes caused by the WS n S to WS2S transformation can now also be given separate state spaces, causing yet a degree of factorization.

One consequence is that type well-formedness now can be represented by a GTA with a constant number of states in each state space. The size of this automaton is thus reduced from quadratic to linear in the size of the type. Similar improvements are observed for other predicates.

With these obstacles removed, implementation becomes feasible with typical data-type operations verified in seconds. In fact, for the linear sub-language, our new decision procedure is almost as fast as the previous WS1S implementation; for example, the programs *reverse* and *zip* from [120] are now verified in 2.3 and 29 seconds instead of the previous times of 2.7 and 10 seconds (all using the newest version of

MONA). This is remarkable, since our decision procedure suffers a quadratic penalty from using tree automata rather than string automata.

7.5 Conclusion

By introducing the WSRT logic and exploiting novel features of the MONA implementation, we have built a tool that catches pointer errors in programs working on recursive data structures. Together with assisting tools for extracting counterexamples and graphical program simulations, this forms the basis for a compile-time debugger that is sound and furthermore complete for loop-free code. The inherent non-elementary lower bound of WS n S will always limit its applicability, but we have shown that it handles some realistic examples.

Among the possible extensions or variations of the technique are allowing parent and root pointers in all structures, following the ideas from [138], and switching to a finer store granularity to permit casts and pointer arithmetic. A future implementation will test these ideas. Also, it would be interesting to perform a more detailed comparison of the technique presented here with pointer analysis and shape analysis techniques.

Chapter 8

The Pointer Assertion Logic Engine

with Michael I. Schwartzbach

Abstract

We present a new framework for verifying partial specifications of programs in order to catch type and memory errors and check data structure invariants. Our technique can verify a large class of data structures, namely all those that can be expressed as *graph types*. Earlier versions were restricted to simple special cases such as lists or trees. Even so, our current implementation is as fast as the previous specialized tools.

Programs are annotated with partial specifications expressed in Pointer Assertion Logic, a new notation for expressing properties of the program store. We work in the logical tradition by encoding the programs and partial specifications as formulas in monadic second-order logic. Validity of these formulas is checked by the MONA tool, which also can provide explicit counterexamples to invalid formulas.

To make verification decidable, the technique requires explicit loop and function call invariants. In return, the technique is highly modular: every statement of a given program is analyzed only once.

The main target applications are safety-critical data-type algorithms, where the cost of annotating a program with invariants is justified by the value of being able to automatically verify complex properties of the program.

8.1 Introduction

We present a new contribution to the area of *pointer verification*, which is concerned with verifying partial specifications of programs that make explicit use of pointers. In practice, there is an emphasis on catching type and memory errors and checking data structure invariants.

For data-type implementations, standard type-checking systems, as in C or Java, are not sufficiently expressive. For example, the type of binary trees is identical to the one for doubly-linked lists. Both are just records with pairs of pointers, which makes the type checker fail to catch many common bugs. In contrast, pointer verification can

validate the underlying data structure invariants, for instance, to guarantee that doubly-linked lists maintain their shapes after pointer manipulations. Memory errors, such as dereference of null pointers or dangling references, and creation of memory leaks are also beyond the scope of standard type checking.

There have been several different approaches to pointer verification, but not many that are as expressive as the one we propose in this paper. Clearly there is a trade-off between expressiveness and complexity, since less detailed analyses will be able to handle larger programs. Our approach is designed to verify a single abstract data type at a time. Since such implementations often contain intricate pointer manipulations and are trusted implicitly by programmers, they are a fair target for detailed scrutiny.

We work in the logical tradition by encoding programs and partial specifications as formulas in monadic second-order logic. Formulas are processed by the MONA tool [131, 158] which reduces them to equivalent tree automata from which it is simple to conclude validity or to extract concrete counterexamples. Translated back into the underlying programming language, a counterexample is an initial store that causes the given program to fail. Program annotations, in the form of assertions and invariants, are allowed and may prove necessary to obtain the desired degree of precision. This approach can be viewed both as lightweight program verification, since the full behavior of the program is not considered, and as heavyweight type checking, since properties well beyond the expressiveness of standard type systems can be checked.

We have reported on our approach in two earlier works. In the first we introduce the basic technique applied to linear lists [120]. In the second we provide a generalization to tree-shaped data structures and introduce a new encoding to make the analysis feasible [82]. The current paper takes a leap forward in generalizing the class of data structures that can be considered, without sacrificing precision or efficiency. Our new framework can handle all data structures that can be described as *graph types* [138]. These include data structures that are well-known from folklore or literature, such as doubly-linked lists, trees with parent pointers, threaded trees, two-dimensional range trees, and endless customized versions such as trees in which all leaves are linked in a cyclic list. Our framework is also designed to handle the common situation where a data structure invariant must be temporarily violated at some program points.

Our contributions are:

- An extension of the results in [120, 82] to the whole class of graph types;
- a language for expressing data structures and operations along with correctness specifications;
- a full implementation exploiting intricate parts of the MONA tool to obtain an efficient decision procedure, together with a range of non-trivial examples.

To verify a data type implementation, the desired data structure is specified in an abstract notation, and the program is annotated with assumptions and assertions. It is not necessary to customize or optimize the implementation, and no proof obligations are left to be dealt with manually.

We rely on a new formal notation, *Pointer Assertion Logic (PAL)* to specify the structural invariants of graph types, to state pre- and post-conditions for procedures, and to formulate invariants and assertions that are given as hints to the system. The

PAL notation is essentially a monadic second-order logic in which the universe of discourse contains records, pointers, and booleans. Programs with PAL annotations are verified with the tool PALE, the *Pointer Assertion Logic Engine*. The “secret” behind the PALE implementation is using the MONA tool to decide validity of Hoare triples based on PAL over loop-free code. Code with loops or recursion is handled by splitting it into loop-free fragments using invariants, as in classical Hoare logic. While the MONA logic has an inherent non-elementary complexity [156], we demonstrate that it can efficiently handle many real programs. Furthermore, the ability to insert assertions to break larger triples into smaller ones suggests that the overall approach is modular and thus can scale reasonably.

A framework for pointer verification, such as ours, should be evaluated on four different criteria. First, how precise is the analysis? Second, it is fast and scalable? Third, does it allow or require programs to be annotated? Fourth, which data structures can be considered and how are they described? In the following sections, we will describe a programming language that uses Pointer Assertion Logic for expression of store properties, describe the decision procedure based on Hoare logic and MONA, and through a number of experiments argue that the Pointer Assertion Logic approach provides a productive compromise between expressibility and efficiency.

8.1.1 A Tiny Example

Consider the type of linked lists with tail pointers, which as a graph type is expressed as:

```
type Head = {
  data first: Node;
  pointer last:
    Node[this.first<next*. [pos.next=null]>last];
}
type Node = {
  data next: Node;
}
```

The notation is explained in the following section, but intuitively the `last` pointer is annotated with a formula that constrains its destination to be the last Node in the list. A candidate for verification is the following procedure which concatenates two such structures:

```
proc concat(data l1,l2: Head): Head
{
  if (l1.last!=null) { l1.last.next = l2.first; }
  else { l1.first = l2.first; }
  if (l2.first!=null) { l1.last = l2.last; }
  l2.first = null;
  l2.last = null;
  return l1;
}
```

These are tedious pointer manipulations that are easy to get wrong. However, if we annotate the procedure with the pre-condition that `l1` and `l2` are not null and run

PALE, it will in half a second report that no memory errors occur and, importantly, that the data structure invariant is maintained.

8.1.2 Related Work

General theorem provers, such as HOL [26], may consider the full behavior of programs but are often slow and not fully automated. Tools such as ESC [74] and LCLint [85] consider memory errors among other undesirable behaviors but usually ignore data structure invariants or only support a few predefined properties. Also, they trade soundness or completeness for efficiency and hence may flag false errors or miss actual errors.

Model checkers such as Bebop [13] and Bandera [65] abstract away the heap and only verify properties of control flow. The JPF [103] model checker verifies simple assertions for a subset of Java, but does not consider structural invariants.

The constraint solver Alloy has been used to verify properties about bounded initial segments of computation sequences [118]. While this is not a complete decision procedure even for straight-line code, it finds many errors and can produce counterexamples. With this technique, data structure invariants can be expressed in first-order logic with transitive closure. However, since it assumes computation bounds, absence of error reports does not imply a guarantee of correctness, and the technique does not appear to scale.

The symbolic executor PREFIX [52] simulates unannotated code through possible executions paths and detects a large class of errors. Again, this is not a complete or sound decision procedure, and data structure invariants are not considered. However, PREFIX gives useful results on huge source programs.

Verification based on static analysis has culminated with shape analysis. The goals of the shape analyzer TVLA [148, 193, 147] are closest to ours but its approach is radically different. Rather than encoding programs in logic, TVLA performs fixed-point iterations on abstract descriptions of the store. Regarding precision and speed, PALE and TVLA seem to be at the same level. TVLA can handle some data abstractions and hence reason about sorting algorithms; we show in Section 8.6 that we can do the same. TVLA analyzes programs with only pre- and post-conditions, where PALE often uses loop invariants and assertions. This seems like an undisputed advantage for TVLA; however, not having invariants can cause a loss in precision making TVLA falsely reject a program. Regarding the specification of new data structures we claim an advantage. Once a graph type has been abstractly described with PAL, the PALE tool is ready to analyze programs. In TVLA it is necessary to specify in three-valued logic an operational semantics for a collection of primitive actions specific to the data structure in question. Furthermore, to guarantee soundness of the analysis, this semantics should be proven correct by hand. TVLA is applicable also to data structures that are not graph types, but so far all their examples have been in that class. Unlike PALE, TVLA cannot produce explicit counterexamples when programs fail to verify.

There exists a variety of assertion languages designed to express properties of data structures, such as ADDS [104], L_r [21], and Shape Types [91]. We rely on PAL since it provides a high degree of expressiveness while still having a decision procedure that works in practice.

A drawback of our approach is that detailed, explicitly stated loop invariants often are required. The overhead of adding such annotations can be significant, so the approach is not applicable for verifying large programs. However, the most complex pointer operations often occur in *data-type implementations*, which usually have a manageable size and appear in central libraries. Thus, we primarily aim for the niche of safety-critical data-type implementations. For such programs, it is well known that the effort of constructing loop invariants is comparable to the effort of designing the data-type [100]. Once the program annotations have been added, the PALE tool can automatically decide validity. PALE works by splitting the program into disjoint fragments that are verified separately by analyzing every statement exactly once. That is, verification depends only on locally specified properties and there is no fixed-point iteration involved. In this sense, the approach is highly scalable. On the other hand, the approach relies on a decision procedure with a non-elementary complexity, so there are programs that cannot be verified in practice. The experiments described in Section 8.7 indicate that the annotation overhead is manageable, that the theoretical complexity is not necessarily a problem in practice, and that quite intricate properties can be expressed and verified.

8.2 Pointer Assertion Logic

In this section, we informally present the components of our framework. First, we describe the underlying *store model*. Second, we use the notion of *graph types* to describe data structures. Third, we employ a simple *programming language* to express data structure operations. And, finally, we use *program annotations* in the form of Pointer Assertion Logic formulas, for expressing properties of the program store.

The programming language and the annotations have been designed to be simple but at the same time as expressive as the verification technique allows. In the following, we present the framework informally and refer the reader to [159] for formal definitions. To make the expressive power of the framework lucid, we show the complete syntax instead of only describing the main ideas.

8.2.1 Store Model

In our model, the store consists of a *heap* and some *program variables*. The heap contains *records* whose *fields* are either *pointers* or *boolean* values. A pointer either has the value `null` or points to a record. Program variables are either *data variables* or *pointer variables*. A data variable is the root of a data structure, whereas a pointer variable may point to any record in the heap.

This is a very concrete representation. We only abstract away arithmetic values and the actual addresses of records. Memory management is not automatically represented, but as in [120, 82], allocation and deallocation primitives could easily be added along with automatic checks for memory leaks and dangling references.

8.2.2 Graph Types

Collections of records and pointers can form any number of interesting data structures, which are generally expressed through an invariant on the allowed shapes. We wish

to explicitly declare such data structures so that their invariants can be verified by our system. For this purpose, we use *graph types* [138] which is an intuitive notation that makes it feasible to describe complex structures. Invariants of graph type structures can be expressed in monadic second-order logic on finite trees, which allows us to use the MONA tool to verify correctness.

A graph type is a tree-shaped data structure with extra pointers. The underlying tree is called the *backbone*. The constituent records have two kinds of fields: *data fields* which define the backbone, and *pointer fields* which may point anywhere in the backbone. To describe a structural invariant, a pointer field is annotated with a *routing expression* which restricts its destination. In the current work, we have generalized the annotations to be arbitrary formulas that may contain routing expressions as basic predicates. Another difference to [138] is that instead of building types from unions and records, we only use records and nullable pointers. Clearly, the two variations can encode each other; we choose the more primitive version, since it turns out to lead to a more efficient decision procedure. Our syntax and semantics of graph type declarations is described in the next section.

Surprisingly many data structures can be described as graph types. As a simple example, consider the type of binary trees where all nodes contain pointers to the root. In our notation, it looks like:

```
type Tree = {
  data left,right:Tree;
  pointer root:Tree[root<(left+right)*>this &
    empty(root^Tree.left union
      root^Tree.right)];
}
```

The syntax for formulas is presented below, but the restriction on the source, *this*, and destination, *root*, of the pointer is read as follows: *this* must be reachable from *root* by following a sequence of *left* or *right* pointers, and the set of *Tree* records having *left* or *right* pointers to the *root* must be empty. Another example is doubly-linked lists with boolean values:

```
type Node = {
  bool value;
  data next:Node;
  pointer prev:Node[this^Node.next={prev}];
}
```

Here, the set of nodes that can reach the *this* node through a *next* pointer must only contain the *prev* node. The convention that *{null}* is interpreted as the empty set handles the first node in the list.

Our benchmark programs cover a variety of data structures expressed as graph types, including singly-linked lists, doubly-linked lists with tail pointers, red-black search trees, and post-order threaded trees with parent pointers. Additional examples are presented in [138].

8.2.3 The Programming Language

A *program* consists of a set of declarations of types, variables, and procedures, specified by the following grammar:

```

typedekl  →  type  $T = \{ ( \text{field} ; )^* \}$ 
field      →  data  $p^\oplus : T$ 
           |  pointer  $p^\oplus : T [ \text{form} ]$ 
           |  bool  $b^\oplus$ 
progv      →  data  $p^\oplus : T$ 
           |  pointer  $p^\oplus : T$ 
           |  bool  $b^\oplus$ 
procedure →  proc  $n ( \text{progv}^\otimes ) : ( T \mid \text{void} )$ 
           |  (  $\text{logicvar} ; )^*$ 
           |   $\text{property}$ 
           |  (  $\{ ( \text{progv} ; )^* \text{stm} \} )^?$ 
           |   $\text{property}$ 

```

We use the notation \oplus and \otimes for comma-separated lists with one-or-more elements and zero-or-more elements, respectively. T , p , b , and n range over names of types, pointer variables or fields, boolean variables or fields, and procedures, respectively. Ignore for now all occurrences of *logicvar* and *property*; they are introduced later. A type consists of a number of fields of kind *data*, *pointer*, or *bool*. The *data* fields span the tree value and the *pointer* fields define extra pointers whose destinations are constrained by a formula. The *bool* fields are used to model finite values. A procedure has a name, formal parameters, a return type, and a body consisting of local variable declarations and statements. If the body is omitted, the declaration is considered a *prototype*.

A statement is one of the following constructs; the *assert* and *split* statements are described later:

```

stm  →  stm stm
      |   $\text{asn}^\oplus ;$ 
      |   $\text{proccall} ;$ 
      |   $\text{if } ( \text{condexp} ) \{ \text{stm} \} ( \text{else } \{ \text{stm} \} )^?$ 
      |   $\text{while } \text{property } ( \text{condexp} ) \{ \text{stm} \}$ 
      |   $\text{return } \text{progexp} ;$ 
      |   $\text{assert } \text{property} ;$ 
      |   $\text{split } \text{property } \text{property} ;$ 
asn  →   $\text{lbexp} = ( \text{condexp} \mid \text{proccall} )$ 
      |   $\text{lptrex} = ( \text{ptrex} \mid \text{proccall} )$ 

```

The language permits multiple-assignment statements where all right-hand sides are evaluated before assigning—these are useful for certain program transformations. Expressions have the following form:

```

condexp →  bexp  |  ?  |  [ form ]
bexp     →  ( bexp )  |  ! bexp
           |  bexp & bexp  |  bexp | bexp
           |  bexp => bexp  |  bexp <=> bexp

```

		$bexp = bexp$		$ptrexp = ptrexp$
		$bexp \neq bexp$		$ptrexp \neq ptrexp$
		$true$ $false$		$lbexp$
$lbexp$	\rightarrow	b $ptrexp . b$		
$ptrexp$	\rightarrow	$null$ $lptrexp$		
$lptrexp$	\rightarrow	p $ptrexp . p$		
$proccall$	\rightarrow	$n ((condexp \mid ptrexp)^{\oplus}) [formula]$		

The “?” operator stands for nondeterministic boolean choice, which is used to model arithmetic conditions that we cannot capture precisely. The operator “.” dereferences a pointer, and the other constructs have the expected meanings.

The language does not contain arithmetic, since our approach focuses on the structural aspects of data types. However, as described in a later section, the technique does permit abstractions of arithmetic properties, for instance for specifying certain ordered data structures.

8.2.4 Program Annotations

Pointer Assertion Logic is a *monadic second-order logic on graph types*. It allows quantification over heap records, both of individual elements and of sets of elements, and uses generalized routing expressions [138] for convenient navigation in the heap. Formulas are used in pointer fields to constrain their destinations, in while loops and procedure calls as invariants, in procedure declarations as pre- and post-conditions, and in assert and split statements. The syntax of formulas is as follows:

$form$	\rightarrow	$(existpos \mid allpos) p^{\oplus} \text{ of } T : form$
		$(existset \mid allset) s^{\oplus} \text{ of } T : form$
		$(existptr \mid allptr) p^{\oplus} \text{ of } T : form$
		$(existbool \mid allbool) s^{\oplus} : form$
		$(form) \quad \mid \quad ! form$
		$form \ \& \ form \quad \mid \quad form \ \mid \ form$
		$form \Rightarrow form \quad \mid \quad form \Leftrightarrow form$
		$ptrexp \text{ in } setexp \quad \mid \quad setexp \text{ sub } setexp$
		$setexp = setexp \quad \mid \quad setexp \neq setexp$
		$empty (setexp) \quad \mid \quad bexp$
		$return \quad \mid \quad n . b$
		$m ((form \mid ptrexp \mid setexp)^{\oplus})$
		$ptrexp < routingexp > ptrexp$
$predicate$	\rightarrow	$pred \ m \ (\ logicvar^{\oplus}) = form$

The identifiers m and s denote predicates and set variables, respectively. The pos and ptr quantifiers differ in that the former range over heap records while the latter also includes the null value. A routing expression formula $p_1 <r> p_2$ is satisfied by a given model if there is a path from p_1 to p_2 satisfying r , as defined below. For reuse of formulas, predicates can be defined as top-level declarations.

Logical variables can be associated to procedures to allow the pre- and post-conditions to be related, as commonly seen in the literature [100, 67]. A logical variable is a universally quantified variable that may occur in the pre- and post-conditions of a procedure but not in the procedure body:

$$\begin{array}{lcl} \text{logicvar} & \rightarrow & \text{pointer } p^{\oplus} : T \\ & | & \text{bool } b^{\oplus} \\ & | & \text{set } s^{\oplus} : T \end{array}$$

In formulas, *ptrexp* has two additional forms allowing access in procedure post-condition to the returned value and in procedure call formulas to the logical variables of the called procedure:

$$\text{ptrexp} \rightarrow \dots \mid \text{return} \mid n.p$$

Set expressions can contain the usual set operators, along with the *up* operation $x \hat{T}.p$ which denotes the set of records of type T having a p successor to x :

$$\begin{array}{lcl} \text{setexp} & \rightarrow & s \\ & | & \text{ptrexp} \hat{T}.p \\ & | & \{ \text{ptrexp}^{\oplus} \} \\ & | & \text{setexp union setexp} \\ & | & \text{setexp inter setexp} \\ & | & \text{setexp minus setexp} \end{array}$$

The syntax of routing expressions is a slightly generalized version of that in [138]. A routing expression is a regular expression over routing directives, each being a step *down* or *up* a pointer or data field, or a formula with the extra free variable *pos* filtering away those records that cause the formula to evaluate to false when *pos* denotes one of them:

$$\begin{array}{lcl} \text{routingexp} & \rightarrow & p \mid \hat{T}.p \mid [\text{form}] \\ & | & \text{routingexp} . \text{routingexp} \\ & | & \text{routingexp} + \text{routingexp} \\ & | & (\text{routingexp}) \mid \text{routingexp} * \end{array}$$

By default, a pointer field must satisfy the formula given in its type declaration. This can be overridden with *pointer directives* of the form:

$$\text{ptrdirs} \rightarrow \{ (T.p [\text{form}])^{\oplus} \}$$

They allows pointer fields to be constrained differently at different program points. This is important because temporary but intentional invalidation of data structure invariants often occurs in imperative programs, as noted for instance in [104]. Pointer directives, both default and overriding, are required to be *well-formed*. This means that in any store and for any record, the directives associated to the pointer fields must denote *exactly one* record. Fortunately, as proved in [138] this is decidable.

A pair consisting of a formula and a set of pointer directives:

$$\text{property} \rightarrow [\text{form ptrdirs}]$$

is called a *property* and denotes the set of stores where

- the formula *form* is satisfied;
- the data variables denote disjoint acyclic backbones spanning the heap; and

- each `pointer` field satisfies its pointer directive (which is either the default from the type declaration or the overriding from the *ptrdirs*).

Properties occur as procedure *pre-* and *post-conditions*, as *while loop invariants*, as *split assertions* and *assumptions* (*split* contains two properties), and as *assert assertions*.

8.2.5 Semantics of Annotations

The program annotations are invariants of the program that must be interpreted as follows:

- The pre-condition of a procedure may be assumed to hold when evaluating the procedure body;
- the post-condition must hold upon termination of the procedure body;
- every *while* loop invariant must hold upon entry and after each iteration, and may be assumed to hold when the loop terminates;
- assertions specified with `assert` must hold at those program points;
- for *split* statements, the assertion properties must hold, and the assumption properties may be assumed to hold (the reason for introducing these statements is explained in Section 8.4); and
- at every procedure call, the invariant conjoined with the pre-condition of the called procedure must hold for some valuation of its logical variables, and the invariant conjoined with the post-condition may be assumed upon return, also for some valuation of the logical variables.

In later sections, we show that the requirements imposed by the annotations can be verified automatically, provided that valid and sufficiently detailed invariants are given.

8.3 Example: Threaded Trees

Before describing our decision procedure, we show a larger example of using PAL. A *threaded tree* is a binary tree in which all nodes contain a pointer to its cyclic successor in a post-order traversal. As a further complication, we equip all nodes with a parent pointer as well. This corresponds to the following graph type:

```
type Node = {
  data left, right: Node;
  pointer post: Node[POST(this, post)];
  pointer parent: Node[PARENT(this, parent)];
}
```

where `POST` and `PARENT` are predicates that spell out these relationships. For example, `PARENT(a, b)` abbreviates the formula:

```
a^Node.left union a^Node.right={b}
```

The POST predicate is more involved and makes use of auxiliary predicates LEAF, ROOT, and LESSEQ.

We consider a procedure `fix(x)` that assigns the correct value to `x.post` assuming that this field initially contains the value `null` and that `x` is non-null. This is a non-trivial operation that looks like:

```
proc fix(pointer x: Node): void
{
  if (x.left=null & x.right=null) {
    if (x.parent=null) { x.post = x; }
    else {
      if (x.parent.right=null | x.parent.right=x) {
        x.post = x.parent;
      }
      else {
        x.post = findsmallest(x.parent.right);
      }
    }
  }
  else { x.post = findsmallest(x); }
}
```

where the auxiliary procedure `findsmallest` is:

```
proc findsmallest(pointer t: Node): Node
  pointer T: Node;
{
  while (t.left!=null | t.right!=null) {
    if (t.left!=null) { t = t.left; }
    else { t = t.right; }
  }
  return t;
}
```

The question is: Does this code verify? Does the resulting tree always satisfy the data structure invariant? Can type or memory errors ever occur? PALE can provide the answers with some help from us. First, since the argument to `fix` is not a proper threaded tree, we must state a suitable pre-condition as the property:

```
[x!=null {Node.post[ALMOSTPOST(this,post,x)]]]
```

Here we require that the argument is not `null` and that the data structure invariant can be temporarily violated. The `ALMOSTPOST` predicate is:

```
(this!=x => POST(this,post)) & (this=x => post=null)
```

which simply states the exception that we allow. Second, the `while` loop in `findsmallest` needs an invariant, which is the property:

```
[INV {Node.post[ALMOSTPOST(this,post,x)]]]
```

where the pointer directive states that the threaded tree is still messed up, and the proper invariant *INV* equals:

$$T < (\text{left} + \text{right}) * > t \ \& \\ \text{allpos } c \text{ of Node: LESSEQ}(c, t, T) \Rightarrow t < (\text{left} + \text{right}) * > c$$

which states that *t* is a descendant of *T* and all its post-order successors are further descendants. See [159] for the full code with all post-conditions. In total, six annotations are required. In less than 4 seconds PALE verifies that the code contains no errors.

8.4 Hoare Logic Revisited

Given an annotated program, we wish to decide whether the program is correct with respect to the annotations. The first step in our decision procedure is to split the given program into Hoare triples [106, 5, 67]. The idea of modeling transformations of the heap with Hoare logic has been studied before [189, 90]. The main novelty of our approach is the choice of PAL as assertion language. Our Hoare “triples” have a non-standard form:

$$\text{triple} \rightarrow \text{property } \text{stm}$$

The statement *stm* is not allowed to contain `while` loops, `split` statements, or procedure calls. A triple is *valid* if

- executing *stm* in a store where *property* is satisfied cannot violate any assertions specified by `assert` statements occurring in *stm*; and
- the execution always terminates in a store consisting of disjoint, acyclic backbones spanning the heap in which all pointer directives hold.

As opposed to normal Hoare triples, these have no explicit post-condition, but the *stm* part may contain `assert` sub-statements. This simple generalization allows many assertions to be made without always breaking triples into smaller parts, as was often the case in [120] and [82]. For instance, an `if` statement where both branches end in `assert` statements does not necessarily need to be broken into two parts. Also, using this form of Hoare triples simplifies the encoding in monadic second-order logic described in Section 8.5.

We define the *cut-points* of a program (according to [88]) as the following set of program points: the beginning and end of procedure bodies and `while` bodies, the `split` statements (these do not affect the computation and are considered single program points), and before each procedure call.

For each cut-point in the given program, we generate a Hoare triple from the property associated with that point and the code that follows until reaching other cut-points. Extra `assert` statements are automatically inserted for these other cut-points, reflecting the assertions they define. In case of `split` statements, we here use the assertion property. For procedure calls, we use the pre-condition property of the called procedure conjoined with the call invariant formula. Recall that we do allow `if` statements

in the Hoare triples. However, if one branch contains a cut-point, we require syntactically that the other branch also contains a cut-point or that the `if` statement is immediately followed by one. Typically, `split` statements are used to fulfill this requirement. As a result, the statement part of a Hoare triple in general has a tree shape with one cut-point in the root and one in each leaf. See [159] for more details.

We claim without proof that this reduction is semantically sound, with two exceptions:

- For `split` statements, the assertion property may not be implied by the assumption property, thereby causing a “gap” between the Hoare triples. This is intentional, because it allows to recover from situations where the required properties are beyond what is expressible in Pointer Assertion Logic, such as arithmetical properties. Using `split` statements at a few selected places, one can then still verify properties of the remaining parts of the code. However, none of the examples shown in Section 8.7 require this feature.
- Procedure calls are known to cause complications for Hoare logic [67]. In our case, there is in general no guarantee that the call invariant is actually a valid invariant. However, in most situations, simple syntactic requirements suffice, since recursive calls in data type operations typically follow the recursive structure of the graph type backbones. A sufficient condition is that the call invariant only accesses variables and record fields that are not assigned to in the procedure. Such requirements ensure that the invariant and the procedure’s pre- and post-conditions express properties of disjoint parts of the store, reminiscent of the “independent conjunctions” in [189]. All the examples shown in Section 8.7 can be handled by simple rules, which we plan to build into PALE.

In PALE, this phase is implemented as a desugaring process reducing all procedures, while loops, `split` statements, and procedure calls to *transduction* declarations having the form “`transduce triple`”. In the following section we describe how validity of these simpler *transduce* constructs can be decided.

In contrast to techniques based on generating the weakest preconditions for all procedures, each program or procedure is not turned into one single verification condition; instead we use the annotations to split the program into Hoare triples that are verified independently. Also, as opposed to [90], we will not rely on fixed-point iterations. This means that detailed invariants may be required; however, it has the advantage that the technique becomes highly modular and hence scalable.

8.5 Deciding Hoare Triples in MONA

We need to decide validity of a Hoare triple of the form

$$\text{property } stm$$

where the statement *stm* is without loops and procedure calls. The question is whether every execution of *stm* starting from a store satisfying *property* is guaranteed to satisfy the assertions given by `assert` statements and to result in stores with disjoint, acyclic backbones spanning the heap in which all relevant pointer directives hold. A

result in [139] shows in a very general setting that this is a decidable question. In essence, we encode each Hoare triple in the logic *weak monadic second-order theory of 2 successors*, which is decidable using the MONA tool [131, 126, 158].

Similarly to the previous implementations [120, 82] we use a particular *transduction* technique. This idea allows us to avoid an explicit construction of weakest pre-conditions working backwards through the statement sequence. Instead, we directly simulate (transduce) the statements and mirror their effect by updating a fixed collection of *store predicates* which abstractly describes a set of stores. It is shown in [139] that any question about the resulting set of stores can be answered by phrasing it in terms of the transduced store predicates and checking for validity of the resulting formula.

The store predicates describe a set of stores in MONA logic. They can be thought of as an interface for asking questions about a store. There are 11 kinds of predicates:

- `bool_T_b(v)` gives the value of the `bool` field `b` in a record `v` of type `T`;
- `succ_T_d(v, w)` holds if the record `w` is reachable from the record `v` of type `T` along a data field named `d`;
- `null_T_d(v)` holds if the data field `d` in the record `v` of type `T` is `null`;
- `succ_T_p(v, w)` holds if the record `w` is reachable from the record `v` of type `T` along a pointer field named `p`;
- `null_T_p(v)` holds if the pointer field `p` in the record `v` of type `T` is `null`;
- `ptr_d(v)` holds if the record `v` is the value of the data variable `d`;
- `null_d()` holds if the data variable `d` is `null`;
- `ptr_p(v)` holds if the record `v` is the destination of the pointer variable `p`;
- `null_p()` holds if the pointer variable `p` is `null`;
- `bool_b()` gives the value of the boolean variable `b`;
- `memfailed()` holds if a null-pointer dereference has occurred.

All properties of a store can be expressed using these predicates in MONA logic. The transduction process generates a collection of such store predicates for each program point. For convenience, we describe this by indexing the predicates with program points; for example, for each program point `i` there is a version of the `bool_T_b(v)` predicate called `bool_T_b_i(v)`.

An initial collection of store predicates is defined to reflect the formula and pointer directives that constitute the pre-condition of the Hoare triple. In the encoding into MONA code, the program variables are modeled as free variables, which are universally quantified in the final validity formula that is given to MONA. For example, a `bool` variable is modeled as a boolean variable `_bool_b` in MONA and the corresponding initial store predicate is:

`bool_b_0() = _bool_b`

Similarly, a pointer variable p is modeled as a first-order MONA variable $_ptr_p$ and the corresponding initial store predicate is:

$$ptr_p_0(v) = v = _ptr_p$$

A `bool` field b in a record of type T is modeled as a second-order variable $_bool_T_b$ containing the set of records in which b is true. Consequently, the corresponding initial store predicate is:

$$bool_T_b_0(v) = v \text{ in } _bool_T_b$$

As a final example, we consider pointer fields whose initial store predicate is:

$$succ_T_p_0(this, p) = f$$

where f is the encoding of the formula associated with the p field of T . If the pre-condition of the Hoare triple contains the pointer directive $T.p[form]$, then that formula is $form$, otherwise the default formula from the type definition is used.

Across a simple statement, two collections of store predicates are related in a manner that reflects the semantics of that statement. Consider for example a type of linked lists:

```
type Node = { data next: Node; }
```

and a simple statement involving two pointer variables of type `Node`:

```
p = q.next;
```

If this statement is enclosed by program points i and j , then the store predicates are updated as follows in MONA code:

```
memfailed_j() = memfailed_i() | null_q_i()
ptr_p_j(v) = ex2 w: ptr_q_i(w) & succ_Node_next_i(w, v)
null_p_j() = ex2 w: ptr_q_i(w) & null_Node_next(w)
```

while the other store predicates remain unchanged. The PALE tool generates such store predicate updates for all Hoare triples and subsequently generates formulas to check the required properties. Between conditionals, routing expressions, and various primitive statements this is a complex translation reminiscent of generating machine code in a compiler. The details can be studied in [159]. The way assignments are handled without losing aliasing information, as in the example above, is essentially the same as in [166].

Checking that an assertion property at a given program point cannot be violated can be expressed by encoding the property using the store predicates associated with the program point together with the pre-condition property encoded with the initial store predicates. There is a strong connection between this transduction technique and the more traditional weakest-precondition technique: if the predicate invocations in the MONA formulas are “unfolded”, one essentially gets the weakest pre-condition. The main advantage of using the “forward” transduction technique instead of a “backward” weakest-precondition technique is an implicit reuse of intermediate results.

Checking that the resulting backbones are disjoint, acyclic, and span the heap is based on formulas for expressing transitive closure. Checking that a pointer directive holds is in [138] shown to be decidable in monadic second-order logic. This result generalizes easily to our extension of graph types, where arbitrary formulas rather than only routing expressions can be used as pointer directives.

The MONA tool transforms the resulting formulas, which can be quite large, into equivalent minimal Guided Tree Automata [23] represented as BDD structures [47], and from that either deduces validity or generates counterexample models. In the latter case, the PALE tool decompiles that model into a program store which causes the program to fail. The use of Guided Tree Automata rather than ordinary tree automata yields an exponential saving by factorizing the state space according to the recursive structure of the graph type backbones. Compared to the WSRT technique used in [82], our choice of describing the backbones as records with pointers rather than as recursive types allow a simpler and more efficient automaton guide to be constructed. Also for efficiency reasons, we compile directly into MONA logic rather than use a more high-level logic, such as FIDO [140].

Note that a collection of store predicates is vaguely similar to the abstract store descriptions employed by TVLA. Consequently, it might seem that we could follow their approach and use a fixed-point process to transduce a `while` loop. However, this is in general not possible, since such fixed-points may require transfinite induction. Hence, we resort to using invariants to break up loops.

This transduction approach introduces no imprecision; it is both sound and complete for individual Hoare triples.

8.6 Data Abstractions

In [193, 147], abstractions of the data contained in the heap records can be tracked by specifying suitable *instrumentation predicates*. As an example, a predicate $dle(x, y)$ is used to represent “the data in x is less than or equal to the data in y ”. To illustrate the power of PAL, we show that a similar approach works for our technique.

As an example, we instrument the ubiquitous linked-list `reverse` example to verify that reversal of a list ordered in increasing order results in a list ordered in decreasing order:

- We associate two boolean fields, `next_dle` and `next_dge`, to the `next` field in the linked-list type, with the intended meaning: `next_dle` is true in a given record if the data in the record denoted by the `next` pointer is certain to be *less than or equal* to the data in the given record – and likewise for `next_dge` with *greater than or equal*.
- Similarly, for each pair of program pointer variables, two boolean variables are added to keep track of the relative order of the records being pointed to. With a subsequent dead-code elimination, a total of three boolean variables suffice.
- For each pointer assignment, the new boolean fields and variables are updated accordingly. For instance,

```
list.next = res;
```

is replaced by the multiple-assignment statement:

```
list.next = res, list.next_dle = res_dle_list;
```

reflecting the change of the next field.

If arbitrary PAL formulas are allowed as right-hand sides of the new assignments, even complex reachability properties can be captured. For this example, simple assignments suffice, though. As in [147], this is also sufficient to verify for instance that `bubblesort` actually sorts the elements.

The intellectual effort needed to update the data abstraction bits seems to be the same as to define the required operational semantics in TVLA. As hinted in the example, some degree of automation is possible for our technique; however, we leave that for future work.

Note that many data structures, in particular variations of search trees, can be abstractly described by associating to every node a few of bits of information summarizing properties of the tree. Those data structures can also be verified using techniques like these.

8.7 Implementation and Evaluation

Our verification technique is implemented in a tool called PALE, the Pointer Assertion Logic Engine. Given an annotated program, PALE checks that:

- the pointer directives are well-formed;
- null pointer dereferences cannot occur;
- at each cut-point that the data variables contain disjoint, acyclic backbones spanning the heap and that the assertions and pointer directives are satisfied;
- all `assert` assertions are valid; and
- all cut-point properties are satisfiable.

There is not necessarily an error in the program if a cut-point property is unsatisfiable, but it usually indicates an error in the specification. As previously mentioned, memory allocation can easily be expressed such that the tool would also check for memory leaks and dangling references.

Using PALE, we have evaluated the technique on a number of examples dealing with a variety of data structures. In all cases, we check for memory errors and possible violations of the data structure invariants:

- *Singly-linked lists* with the operations `reverse`, `search`, `zip`, `delete`, `insert`, and `rotate`. These examples have been scrutinized before [64, 120, 148]. We also include the `concat` operation on lists with tail pointers from Section 8.1. We have tried `bubblesort` as in [147] but with various degrees of abstraction of the data: In `bubblesort_simple`, the record values are abstracted away so only null pointer dereferences are checked for; in `bubblesort_boolean`, the

Example name	Lines of code	Invariants (formulas)	GTA operations	Largest GTA		Time (seconds)	Memory (MB)
				States	BDD nodes		
reverse	16	1	1,109	35	142	0.52	2
search	12	1	853	27	85	0.25	2
zip	33	1	1,753	174	730	4.58	11
delete	22	0	973	73	349	1.36	5
insert	33	0	1,005	103	443	2.66	7
rotate	11	0	590	44	213	0.22	1
concat	24	0	1,056	48	177	0.47	3
bubblesort_simple	43	1	1,477	373	3,289	2.86	18
bubblesort_boolean	43	2	1,737	357	3,922	3.37	12
bubblesort_full	43	2	2,069	373	3,291	4.13	19
orderedreverse	24	1	1,091	29	100	0.46	3
recreverse	15	2	1,019	42	176	0.34	2
doublylinked	72	1	4,163	230	796	9.43	13
leftrotate	30	0	1,489	165	1,550	4.62	7
rightrotate	30	0	1,489	165	1,550	4.68	7
treeinsert	36	1	1,989	137	844	8.27	31
redblackinsert	57	7	4,279	297	2,419	35.04	44
threaded	54	4	3,505	50	248	3.38	7

Figure 8.1: Statistics from PALE experiments.

values are abstracted to booleans which in the post-condition are checked to be properly sorted; and in `bubblesort_full`, the data abstraction technique from Section 8.6 is used as in [147] to conclude that the resulting lists are sorted. We also use data abstractions in `orderedreverse` to show that `reverse` switches the order of a sorted list. Finally, we try `recreverse`, which is a recursive version of `reverse`.

- *Doubly-linked lists with tail pointers* [138] with the operations `delete`, `search`, `insert`, and `concat`.
- *Red-black search trees* [66] with the standard operations `leftrotate`, `rightrotate`, `treeinsert`, and `redblackinsert`. We include the non-arithmetic part of the red-black search tree invariant, that is, that the root is black and red nodes have black children:

```
BLACK(root) &
allpos q of Node: ROOT<(left+right)*>q =>
  (RED(q) => BLACK(q.left) & BLACK(q.right));
```

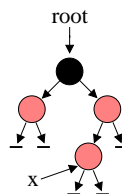
- *Threaded trees* [138], as shown in Section 8.3, where every node has a pointer to its post-order cyclic successor and a pointer to its parent, with a `fix` operation for reestablishing the correct post pointer for a given node.

The resources for translation into MONA code and for the automaton analysis are negligible. Figure 8.1 shows the time and space consumptions of the MONA automaton operations (on a 466MHz Celeron PC) for the examples, along with the number

of GTA operations (here we count only the essential operations: minimization, projection, and product), the size of the largest intermediate minimized automaton (in number of states and in number of BDD nodes). Note that some examples implement individual operations while others implement whole data types. The lines of code measure the underlying program only, thus disregarding the PAL annotations. “Invariants” is the total number of `split` statements, while statements, and procedure calls that require explicitly stated invariants. This number is an indication of the effort required by the programmer to make PALE work, in addition to writing the program and its specification. The invariants for `redblackinsert` were admittedly hard to get right. However, the programs that require the most complicated invariants are also those that have the most complicated pointer operations and hence are the ones in most need of verification. The table shows that the examples typically run in seconds despite requiring a quite large number of automaton operations. Since the complexity is non-elementary in the size of the program, intractable examples do exist but they do not seem to occur often in practice. The verification time seems insignificant compared to the time required to design a given data type and specify the invariants, however, it is useful in the design cycle that verification is efficient.

The code for the bubblesort examples (excluding annotations) is taken from [147]. Interestingly, PALE discovered a minor bug (a null-pointer dereference) even though the code had allegedly been verified by TVLA, which spent 245 seconds compared to 4 seconds for PALE. This huge speedup shows an instance where using invariants is much faster than performing a fixed-point iteration. This suggests that PALE may be quite scalable. Another noteworthy point discovered by PALE is that in [66], the authors forget to require the root to be initially black in `redblackinsert`. (More precisely, they mention the requirement in the proof of correctness, but not in the specification.)

Versions with plausible bugs planted typically take roughly the same time to process as the correct programs. For such buggy versions, counterexamples are generated, which is crucial for determining whether the error is in the program, the assumptions, or the assertions. As an example, if a conditional in `redblackinsert` erroneously tests for a specific node to be black rather than red, PALE produces the following counterexample store for the Hoare triple containing the conditional:



Here, the root node is black and the others are red, and we omit field names and all pointer fields. Such a counterexample is clearly useful for locating the bug. Notice that for this bug, the approach in [118] would not find the bug for heap bounds of less than four records.

The experiments show that our approach does work in practice for non-trivial data structures, and with time and space requirements which are as good as or better than those for the previous more specialized versions [120, 82] and related approaches with similar goals [147, 118, 77, 90].

8.8 Conclusion

It is well known that developing formal program specifications is expensive, but for some safety critical applications a guarantee of partial correctness of data type implementations can be worth the effort. A tool such as PALE can be used to verify specifications expressible in Pointer Assertion Logic, and also to guide the programmer by the generation of counterexamples. With verification techniques based on undecidable logics, either the programmer may have to guide a theorem prover to the proofs, not even being certain that they exist, or accept that the reply may be “don’t-know”. With less expressive techniques, important aspects of the data types may not be expressible and hence not verifiable. In contrast to traditional program analyses, our technique is highly modular: each statement in the given program is analyzed only once. To verify complex properties, the technique often requires detailed invariants to be provided. However, since we primarily aim for data-type implementations, we believe that this annotation overhead is reasonable compared to the effort of creating the program. In conclusion, Pointer Assertion Logic may provide a fruitful compromise between expressibility and usability.

Although facing a non-elementary theoretical complexity, the examples we provide show that logic and automaton based program verification is feasible. Furthermore, we believe that the efficiency of the implementation can be improved by at least an order of magnitude by tuning the MONA tool using heuristics as proposed in [134]. As also suggested in [118, 103] we may benefit from an initial simplification phase that performs program slicing or partial evaluation of the source programs.

Future work will also examine the possibility of incorporating simple arithmetic into the language. The MONA tool can also be used as an efficient decision procedure for Presburger arithmetic [198, 131], which is sufficient for many properties. In [104], abstract data structure descriptions are used to improve program analyses in optimizing compilers. Pointer aliasing, for instance, can be expressed in PAL, so the detailed knowledge of the heap structure provided by PALE might also be useful for optimization. Another idea is to build a translator from C, C++, or Java to PALE to make the tool more practically useful. Finally, it might be interesting to integrate the “independent conjunctions” from [189] into PAL to support local reasoning and make the tool easier to use.

The full source code for the PALE tool, the examples, and a detailed description of the desugaring and code generation to MONA are available from the PALE site at <http://www.brics.dk/PALE/>.

Acknowledgments

This work was done while the first author was visiting UC Berkeley. Thanks to Alex Aiken, Jeff Foster, and Zhendong Zu for valuable discussions and comments.

Chapter 9

The <bigwig> Project

with Claus Brabrand and Michael I. Schwartzbach

Abstract

We present the results of the <bigwig> project, which aims to design and implement a high-level domain-specific language for programming interactive Web services.

A fundamental aspect of the development of the World Wide Web during the last decade is the gradual change from static to dynamic generation of Web pages. Generating Web pages dynamically in dialog with the client has the advantage of providing up-to-date and tailor-made information. The development of systems for constructing such dynamic Web services has emerged as a whole new research area.

The <bigwig> language is designed by analyzing its application domain and identifying fundamental aspects of Web services inspired by problems and solutions in existing Web service development languages. The core of the design consists of a session-centered service model together with a flexible template-based mechanism for dynamic Web page construction. Using specialized program analyses, certain Web-specific properties are verified at compile time, for instance that only valid HTML 4.01 is ever shown to the clients. In addition, the design provides high-level solutions to form field validation, caching of dynamic pages, and temporal-logic based concurrency control, and it proposes syntax macros for making highly domain-specific languages.

The language is implemented via widely available Web technologies, such as Apache on the server-side and JavaScript and Java Applets on the client-side. We conclude with experience and evaluation of the project.

9.1 Introduction

The <bigwig> project was founded in 1998 at the BRICS Research Center at the University of Aarhus to design and implement a high-level domain-specific language for programming interactive Web services. Such services are characterized by involving multiple interactions with each client, mediated by HTML forms in browsers. In the following we argue that existing Web service programming languages in various ways provide only low-level solutions to problems specific to the domain of Web services.

Our overall ambitions for the project are to identify the key areas of the Web service domain, analyze the problems with the existing approaches, and provide high-level solutions that will support development of complex services.

9.1.1 Motivation

Specifically, we will look at the following Web service technologies: the HTTP/CGI Web protocol [101], Sun's Java Servlets [201] and their JavaServer Pages (JSP) [202], Microsoft's Active Server Pages (ASP) [107], the related Open Source language PHP [9], and the research language MAWL [8, 7, 144].

CGI was the first platform for development of Web services, based on the simple idea of letting a script generate the reply to incoming HTTP requests dynamically on the server, rather than returning a static HTML page from a file. Typically, the script is written in the general-purpose scripting language Perl, but any language supported by the server can be used. Being based on general-purpose programming languages, there is no special support for Web specific tasks, such as generation of HTML pages, and knowledge of the low-level details of the HTTP protocol are required. Also, HTTP/CGI is a stateless protocol that by itself provides no help in tracking and guiding users through a series of individual interactions. This can to some degree be alleviated by libraries. In any case, there are no compile-time guarantees of correct runtime behavior when it comes to Web-specific properties, for instance ensuring that invalid HTML is never sent to the clients.

Servlets are a popular higher-level Java-specific approach. Servlets, which are special Java programs, offer the common Java advantages of network support, strong security guarantees, and concurrency control. However, some significant problems still exist. Services programmed with servlets consist of collections of request handlers for individual interactions. Sessions consisting of several interactions with the same client must be carefully encoded with cookies, URL rewriting, or hidden input fields, which is tedious and error-prone even with library support, and it becomes hard to maintain an overview of large services with complex interaction flows. A second, although smaller, problem is that state shared between multiple client sessions, even for simple services, must be explicitly stored in a name-value map called the "servlet context", instead of using Java's standard variable declaration scoping mechanism. Thirdly, the dynamic construction of Web pages is not improved compared to CGI. Web pages are built by printing string fragments to an output stream. There is no guarantee that the result will always become valid HTML. This situation is slightly improved by using HTML constructor libraries, but they preclude the possibility of dividing the work of the programmers and the HTML designers. Furthermore, since client sessions are split into individual interactions that are only combined implicitly, for instance by storing session IDs in cookies, it is not possible to statically analyze that a given page sent to a client always contains exactly the input fields that the next servlet in the session expects.

JSP, ASP, PHP, and the countless homegrown variants were designed from a different starting point. Instead of aiming for complex services where all parts of the pages are dynamically generated, they fit into the niche where pages have mostly static contents and only small fragments are dynamically generated. A service written in one of these languages typically consists of a collection of "server pages" which are HTML

pages with program code embedded in special tags. When such a page is requested by the client, the code is evaluated and replaced by the resulting string. This gives better control over the HTML construction, but it only gives an advantage for simple services where most of every page is static.

The MAWL language was designed especially for the domain of interactive Web services. One innovation of MAWL is to make client sessions explicit in the program logic. Another is the idea of building HTML pages from templates. A MAWL service contains a number of sessions, shared data, and HTML templates. Sessions serve as entry points of client-initiated session threads. Rather than producing a single HTML page and then terminating as CGI scripts or Servlets, each session thread may involve multiple client interactions while maintaining data that is local to that thread. An HTML template in MAWL is an HTML document containing named gaps where either text strings or special lists may be inserted. Each client interaction is performed by inserting appropriate data into the gaps in an HTML template and then sending it to the client, who fills in form fields and submits the reply back to the server.

The notions of sessions and document templates are inherent in the language and, being compilation-based, allow important properties to be verified statically, without actually running the service. Since HTML documents are always constructed from the templates, HTML validity can be verified statically. Also, since it is clear from the service code where execution resumes when a client submits form input, it can be statically checked that the input fields match what the program expects. One practical limitation of the MAWL approach is that the HTML template mechanism is quite restrictive, as we cannot insert markup into the template gaps.

We describe more details about the existing languages in the following sections. By studying services written in any of these languages, some other common problems show up. First of all, often surprisingly large portions of the service code tend to deal with form input validation. Client-server interaction takes place mainly through input forms, and usually some fields must be filled with a certain kind of data, perhaps depending on what has been entered in other fields. If invalid data is submitted, an appropriate error message must be returned so that the client can try again. All this can be handled either on the client-side—typically with JavaScript [87], in the server code or with a combination. In any case, it is tedious to encode.

Second, one drawback of dynamically generated Web pages compared to static ones is that traditional caching techniques do not work well. Browser caches and proxy servers can cause major improvements in saving network bandwidth, load time, and clock cycles, but when moving towards interactive Web services, these benefits disappear.

Third, most Web services act as interfaces to underlying databases that, for instance, contain information about customers, products, and orders. Accessing databases from general-purpose programming languages where database queries are not integrated requires the queries to be built as text strings that are sent to a database engine. This means that there is no static type checking of the queries. As known from modern programming languages, type systems allow many programming bugs to be caught at compile time rather than at runtime, and thereby improve reliability and reduce development cost.

Fourth, since running Web services contain many concurrently executing threads and they access shared information, for instance in databases on the server, there is a

fundamental need for concurrency control. Threads may require exclusive access to critical regions, be blocked until certain events occur, or be required to satisfy more high-level behavioral constraints. All this while the service should run smoothly without deadlocks and other abrupt obstacles. Existing solutions typically provide no or only little support for this, for instance via low-level semaphores as Perl or synchronized methods in Servlets. This can make it difficult to guarantee correct concurrent execution of entire services.

Finally, since Web services usually operate on the Internet rather than on secure local networks, it is important to protect sensitive information both from hostile attacks and from programming leaks. A big step forward is the Secure Sockets Layer (SSL) protocol [93] combined with HTTP Authentication [22]. These techniques can ensure communication authenticity and confidentiality, but using them properly requires insight into technical protocol and implementation details. Furthermore, they do not protect against programming bugs that unintentionally leak secret information. The “taint mode” in Perl offers some solution to this. However, it is runtime based so no compile-time guarantees are given. Also, it only checks for certain predefined properties, and more specialized properties cannot be added.

9.1.2 The <bigwig> Language

Motivated by the languages and problems described above, we have identified the following areas as key aspects of Web service development:

- *sessions*: the underlying paradigm of interactive Web services;
- *dynamic documents*: HTML pages must be constructed in a flexible, efficient, and safe fashion;
- *concurrency control*: Web services consist of collections of processes running concurrently and sharing resources;
- *form field validation*: validating user input requires too much attention from Web programmers so a higher-level solution is desirable;
- *database integration*: the core of a Web service is often a database with a number of sessions providing Web access; and
- *security*: to ensure authenticity and confidentiality, regarding both malicious clients and programming bugs.

To attack the problems, we have designed from scratch a new language called <bigwig>, as a descendant of the MAWL language. This language is a high-level, domain-specific language [214], meaning that it employs special syntax and constructs that are tailored to fit its particular application domain and allow specialized program analyses, in contrast to library-based solutions. Its core is a C or Java-like skeleton, which is surrounded by domain-specific sub-languages covering the above key aspects. A notion of *syntax macros* tie the sub-languages together and provide additional layers of abstraction. This macro language, which operates on the parse tree level, rather than the token sequence level as conventional macro languages, has proved successful in providing extensions of the core language. This has helped each of the sub-languages

remain minimal, since desired syntactic sugar is given by the macros. Syntax macros can be taken to the extreme, where they, with little effort, can define a completely new syntax for *very*-domain-specific languages tailored to highly specialized application domains.

It is important that `<bigwig>` is based on compilation rather than on interpretation of a scripting language. Unlike many other approaches, we can then apply type systems and static analysis to catch many classes of errors before the service is actually installed.

The `<bigwig>` compiler uses common Web technologies as target languages. This includes HTML [185], HTTP [22], JavaScript [87], and Java Applets [6]. Our current implementation additionally relies on the Apache Web server. It is important to apply only standard technologies on the client-side in order not to place restrictions on the clients. In particular, we do not use browser plug-ins, and we only use the subset of JavaScript that works on all common browsers. As new technologies become standard, the compiler will merely obtain corresponding opportunities for generating better code. To the degree possible, we attempt to hide the low-level technical details of the underlying technologies.

We have made no effort to contribute to the graphical design of Web services. Rather, we provide a clean separation between the physical layout of the HTML pages and the logical structure of the service semantics. Thus, we expect that standard HTML authoring tools are used, conceivably by others than the Web programmer. Also, we do not focus on efficiency, but on providing higher levels of abstraction for the developers. For now, we regard it as less important to generate solutions that seamlessly scale to thousands of interactions per second, although, of course, scalability is an issue for the design.

The main contributions of the `<bigwig>` project are the following results:

- The notion of client sessions can and should be made explicit in Web service programming languages;
- dynamic construction of Web pages can at the same time be made flexible and fast, while still permitting powerful compile-time analyses;
- form field validation can be made easier with a domain-specific language based on regular expressions and boolean logic;
- temporal logic is a useful formalisms for expressing concurrency constraints and synthesizing safety controllers; and
- syntax macros can be used to create very-domain-specific high-level languages for extremely narrow application domains.

We focus on these key contributions in the remainder of this article, but also describe less central contributions, such as a technique for performing client-side caching of dynamically generated pages, a built-in relational database, and simple security mechanisms. The individual results have been published in previous more specialized articles [194, 195, 38, 37, 39, 36, 41]. Together, these results show that there is a need for high-level programming languages that are tailor-made to the domain of Web service development.

9.1.3 Overview

We begin in Section 9.2 by classifying the existing Web service languages as script-, page-, or session-centered, arguing for the latter as the best choice for complex services. In Section 9.3, we show how the HTML template mechanism from MAWL can be extended to become more flexible using a notion of higher-order templates. Using novel type systems and static analyses, the safety benefits of MAWL templates remain in spite of the increased expressibility. Also, we show how our solution can be used to cache considerable parts of the dynamically generated pages in the browser. In Section 9.4, we address the problem of validating form input more easily. Section 9.5 describes a technique for generating concurrency controllers from temporal logic specifications. Section 9.6 gives an introduction to the syntax macro mechanism that ties together the sub-languages of <bigwig>. In Section 9.7, we mention various less central aspects of the <bigwig> language. Finally, in Section 9.8 we describe our implementation and a number of applications, and evaluate various practical aspects of <bigwig>.

9.2 Session-Centered Web Services

Web programming covers a wide spectrum of activities, from composing static HTML documents to implementing autonomous agents that roam the Web. We focus in our work on *interactive Web services*, which are Web servers where clients can initiate sessions that involve several exchanges of information mediated by HTML forms. This definition includes large classes of well-known services, such as news services, search engines, software repositories, and bulletin boards, but also covers services with more complex and specialized behavior.

There are a variety of techniques for implementing interactive Web services, but they can be divided into three main paradigms: the *script-centered*, the *page-centered*, and the *session-centered*. Each is supported by various tools and suggests a particular set of concepts inherent in Web services.

9.2.1 The Script-Centered Approach

The script-centered approach builds directly on top of the plain, stateless HTTP/CGI protocol. A Web service is defined by a collection of loosely related scripts. A script is executed upon request from a client, receiving form data as input and producing HTML as output before terminating. Individual requests are tied together by explicitly inserting appropriate links to other scripts in the reply pages.

Perl is a prototypical scripting language, but almost any programming language has been suggested for this role. CGI scripting is often supported by a large collection of library functions for decoding form data, validating input, accessing databases, and realizing semaphores. Even though such libraries are targeted at the domain of Web services, the language itself is not. A major problem is that the overall behavior is distributed over numerous individual scripts and depends on the implicit manner in which they pass control to each other. This design complicates maintenance and precludes any sort of automated global analysis, leaving all errors to be detected in the running service [86, 7].

HTML documents are created on the fly by the scripts, typically using print-like statements. This again means that no static guarantees can be issued about their correctness. Furthermore, the control and presentation of a service are mixed together in the script code, and it is difficult to factor out the work of programmers and HTML designers [68].

The Java Servlets language also fits this category. The overall structure of a service written with servlets is the same as for Perl. Every possible interaction is essentially defined by a separate script, and one must use cookies, hidden input fields, or similar techniques to connect sequences of interactions with the clients. Servlets provide a session tracking API that hides many of the details of cookies, hidden input fields, and URL rewriting. Many servlet servers use cookies if the browser supports them, but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled. This API is exemplified by the following code inspired by two Servlet tutorials:¹

```
public class SessionServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        ServletContext context = getServletContext();
        HttpSession session = request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>Servlet Demo</TITLE></HEAD><BODY>");
        if (session.isNew()) {
            out.println("<FORM ACTION=SessionServlet>" +
                       "Enter your name: <INPUT NAME=handle>" +
                       "<INPUT TYPE=SUBMIT></FORM>");
            session.putValue("state", "1");
        } else {
            String state = (String) session.getValue("state");
            if (state.equals("1")) {
                String name = (String) request.getParameter("handle");
                int users =
                    ((Integer) context.getAttribute("users")).intValue() + 1;
                context.setAttribute("users", new Integer(users));
                session.putValue("name", name);
                out.println("Hello " + name + ", you are user number " + users);
                session.putValue("state", "2");
            } else /* state.equals("2") */ {
                String name = (String) session.getValue("name");
                out.println("Goodbye " + name);
                session.invalidate();
            }
        }
        out.println("</BODY></HTML>");
    }
}
```

Clients running this service are guided through a series of interactions: first, the service prompts for the client's name, then the name and the total number of invocations

¹<http://www.apl.jhu.edu/~hall/java/Servlet-Tutorial/> and <http://java.sun.com/docs/books/tutorial/servlets/>

is shown, and finally a “goodbye” page is shown. The `ServletContext` object contains information shared among all sessions, while the `HttpSession` object is local to each session. The code is essentially a `switch` statement that branches according to the current interaction. An alternative approach is to make a servlet for each kind of interaction. In spite of the API, we still need to explicitly maintain both the state and the identity of the session.

The model of sessions that is supported by Servlets and other script-centered approaches tends to fit better with “shopping basket applications” where the client browses freely among dynamically generated pages than with complex services that need to impose more strict control on the interactions.

9.2.2 The Page-Centered Approach

The page-centered approach is covered by languages such as ASP, PHP, and JSP, where the dynamic code is embedded in the HTML pages. In a sense, this is the inverse of the script-centered languages where HTML fragments are embedded in the program code. When a client requests a page, a specialized Web server interprets the embedded code, which typically produces additional HTML snippets while accessing a shared database. In the case of JSP, implementations work by compiling each JSP page into a servlet using a simple transformation.

This approach is often beautifully motivated by simple examples, where pages are mainly static and only sporadically contain computed contents. For example, a page that displays the time of day or the number of accesses clearly fits this mold. The following JSP page dynamically inserts the current time together with a title and a user name based on the CGI input parameters:

```
<HTML><HEAD><TITLE>JSP Demo</TITLE></HEAD><BODY>
Hello <%
    String name = request.getParameter("who");
    if (name==null) name = "stranger";
    out.print(name);
%>!
<P>
This page was last updated: <%= new Date() %>
</BODY></HTML>
```

The special `<%...%>` tags contain Java code that is evaluated at the time of the request. As long as the code parts only generate strings without markup, it is easy to statically guarantee that all pages shown are valid HTML and other relevant properties. But as the services become more complex, the page-centered approach tends to converge towards the script-centered one. Instead of a mainly static HTML page with some code inserted, the typical picture is a single large code tag that dynamically computes the entire contents. Thus, the two approaches are closely related, and the page-centered technologies are superior only to the degree in which their scripting languages are better designed.

The ASP and PHP languages are very reminiscent of JSP. ASP is closely tied to Microsoft’s Internet Information Server, although other implementations exist. Instead of being based on Java, it defines a language-independent connection between HTML

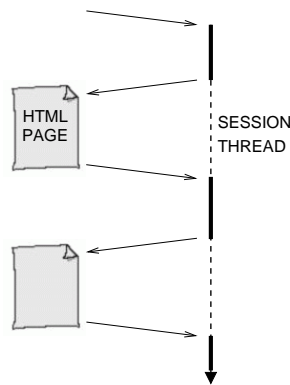


Figure 9.1: Client-server sessions in Web services. On the left is the client's browser, on the right a session thread running on the server. The thread is initiated by a client request and controls the sequence of interactions.

pages and scripting languages, typically either Visual Basic Script or Microsoft's version of JavaScript. PHP is a popular Open Source variant whose scripting language is a mixture of C, Java, and Perl.

These languages generally provide only low-level support for tracking client sessions and maintaining session state. Cookies, hidden input fields, and some library support is the common solution. For other Web service aspects also, such as databases and security, there is often a wide range of libraries available but no direct language support.

9.2.3 The Session-Centered Approach

The pure session-centered approach was pioneered by the MAWL project. Here a service is viewed as a collection of distinct *sessions* that access some shared data. A client may initiate a session *thread*, which is conceptually a process running on the server. Interaction with the client is viewed as remote procedure calls from the server, as known from classical construction of distributed systems but with the roles reversed.

The flow of an entire session is programmed as a single sequential program, which is closer to ordinary programming practice and offers the compiler a chance to obtain a global view of the service. Figure 9.1 illustrates the flow of control in this approach. Important issues such as concurrency control become simpler to understand in this context and standard programming solutions are more likely to be applicable.

The following MAWL program is equivalent to the previous Servlet example:

```
static int users = 0;

session GreetingSession {
  auto form {} -> {handle} hello;
  auto string name = hello.put().handle;

  auto form {string who, int count} -> {} greeting;
  users++;
  greeting.put({name, users});

  auto form {string who} -> {} goodbye;
```

```
    goodbye.put({name});
}
```

The HTML templates *hello*, *greeting*, and *goodbye* are placed in separate files. Here is *hello.mhtml*:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Enter your name: <INPUT NAME=handle>
</BODY></HTML>
```

and *greeting.mhtml*:

```
<HTML><HEAD><TITLE>MAWL Demo</TITLE></HEAD><BODY>
Hello <MVAR NAME=who>, you are user number <MVAR NAME=count>
</BODY></HTML>
```

The template for *goodbye* is similar. A form tag and a continue button are implicitly inserted. Variables declared *static* contain persistent data, while those declared *auto* contain per-session data, also called *local* data. The form variables are declared with two record types. The former defines the set of gaps that occur in the template, and the latter defines the input fields. In the templates, gaps are written with MVAR tags. Template variables all have a *put* method. When this is executed, the arguments are inserted in the gaps, the resulting page is sent to the client who fills in the fields and submits the reply, which is turned into a record value in the program. Note how the notion of sessions is explicit in the program, that private and shared state is simply a matter of variable declaration modifiers, and that the templates are cleanly separated from the service logic. Obviously, the session flow is clearer, both to the programmer and to the compiler, than with the non-session based approaches. One concrete benefit is that it is easy to statically check both validity and correct use of input fields.

The main force of the session-centered approach is for services where the control flow is complex. Many simple Web services are in actuality more loosely structured. If all sessions are tiny and simply do the work of a server module from the page-centered approach, then the overhead associated with sessions may seem too large. Script-centered services can be seen as a subset of the session-centered where every session contains only one client interaction. Clearly, the restriction in the script-centered and the page-centered languages allows significant performance improvements. For instance, J2EE Servlet/JSP servers employ pools of short-lived threads that store only little local state. For more involved services, however, the session-centered approach makes programming easier, since session management comes for free.

9.2.4 Structure of <bigwig> Services

The overall structure of <bigwig> programs is directly inspired by MAWL. A <bigwig> program contains a complete specification of a Web *service*. A service contains a collection of named *sessions*, each of which essentially is an ordinary sequential program. A client has the initiative to invoke a thread of a given session, which is a process on the server that executes the corresponding sequential code and exclusively communicates with the originating client. Communication is performed by *showing* the client an HTML page, which implicitly is made into a form with an appropriate URL return

address. While the client views the given document, the session thread is suspended on the server. Eventually the client submits the form, which causes the session thread to be resumed and any form data entered by the client to be *received* into program variables. A simple <bigwig> service that communicates with a client, as in the Servlet and MAWL examples, is the following:

```
service {
  html hello = <html>Enter your name: <input name=handle></html>;

  html greeting =
    <html>Hello <[who]>, you are user number <[count]></html>;

  html goodbye = <html>Goodbye <[who]></html>;

  shared int users = 0;

  session Hello() {
    string name;
    show hello receive[name=handle];
    users++;
    show greeting<[who=name, count=users]>;
    show goodbye<[who=name]>;
  }
}
```

The program structure is obviously as in MAWL, except that the session code and the templates are wrapped into a *service* block. For instance, the *show-receive* statements produce the client interactions similarly to the *put* methods in MAWL. However, <bigwig> provides a number of new features. Most importantly, HTML templates are now *first-class values*. That is, *html* is a built-in data type, and its values can be passed around and stored in variables as any other data type. Also, the HTML templates are *higher-order*, meaning that instead of only allowing text strings to be inserted into the template gaps, we also allow insertion of other templates. This is done with the special *plug* operator, $x<[y=z]$ which inserts a string or template z into the y gaps of the x template. Clearly, this constitutes a more flexible document construction mechanism, but it also calls for new ideas for statically verifying HTML validity, for instance. This is the topic of Section 9.3. Other new features include the techniques for improving form field validation and concurrency control, together with the syntax macro mechanism, all of which are described in the following sections.

9.2.5 A Session-Based Runtime Model

The session-based model can be implemented on top of the CGI protocol. One naive approach is to create session threads as CGI scripts where all local state is stored on disk. At every session interaction, the thread must be started again and restore its local state, including the call stack, in order to continue execution. A better approach is to implement each session thread as a process that runs for the whole duration of the session. For every interaction, a tiny transient CGI script, called a *connector process*, is executed, acting as a pipe between the Web server and the session process. This approach resembles FastCGI [176], and is described in detail in [38]. Our newest

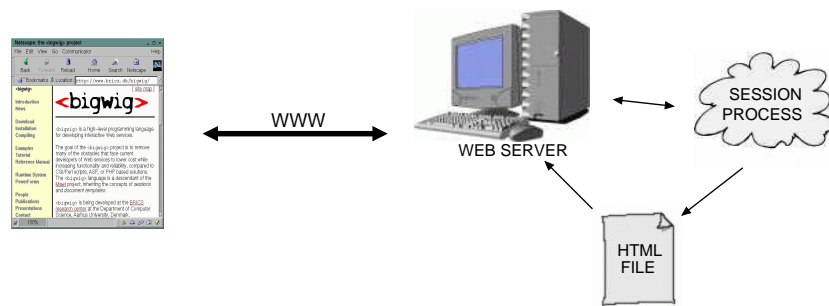


Figure 9.2: Session-based runtime model with reply indirection. Each session thread is implemented as a separate process that writes its HTML reply to a designated file.

implementation is instead based on a specialized Apache server module.² Naturally, this is much faster than the CGI solutions since it does not create a new process for every single interaction, but only for the session processes.

Two common sources of problems with standard implementations of sessions are history buffers and bookmarking features found in most browsers. With history buffers and the “back” button, the users can step back to a previous interaction, and either intentionally or unintentionally resubmit an old input form. Sometimes this can be a useful feature, but more often this causes confusion and annoyance to the users who may, for instance, order something twice. It is a general problem that the information shown to the user in this way can be obsolete since it is tailor-made only for the exact time of the initial request. Since the information was generated from a shared database that may have changed entirely, it does generally not make sense to “step back in time” using the history buffer. This is no different from ordinary programs. Even if the programmer was aware of this and added serial number checks, the history buffer will be full of URLs to obsolete requests. If the service really needs a “back” feature, it can be programmed explicitly into the flow of the sessions. It also becomes hazardous to try to use bookmarks to temporarily suspend a session. Invoking the bookmark will typically cause a CGI script to be executed a second time instead of just displaying its results again.

<bigwig> provides a simple but unique solution to these problems: Each session thread is associated with a URL which points to a file on the server containing the latest HTML page shown to the client. Instead of sending the contents directly to the client at every show statement, we redirect the browser to this URL, as illustrated in Figure 9.2. Since the URL serves as the identification of the session thread, this solves the problems mentioned above: The history list of the browser now only contains a single entry for the duration of the session, the sessions can now be bookmarked for later use, and in addition, the session identity URL can be passed around manually—to another browser, for instance—without problems. When using URLs instead of cookies to represent the session identity, it also becomes possible for a single user to simultaneously run multiple sessions in different windows but with the same browser.

Furthermore, with this simple solution we can automatically provide the client with feedback while the server is processing a request. This is done by, after a few seconds, writing a temporary response to the HTML file, which informs the client

²See <http://httpd.apache.org/>.

about the status of the request. This temporary file reloads itself frequently, allowing for updated status reports. When the final response is ready, it simply overwrites the temporary reply file, causing the reloading to stop and the response to be shown. This simple technique may prevent the client from becoming impatient and abandoning the session.

Additionally, the `<bigwig>` runtime system contains a garbage collector process that monitors the service and shuts down session processes abandoned by the clients. By default, this occurs if the client has not responded within 24 hours. The sessions are allowed to execute some clean-up actions before terminating.

9.3 Dynamic Construction of HTML Pages

In MAWL, all HTML templates are placed in separate files and viewed as procedures of a kind, with the arguments being strings that are plugged into gaps in the template and the results being the values of the form fields that the template contains. This allows a complete separation of the service code and the HTML code. Two benefits are that static guarantees are possible and that the work of programmers and HTML designers can be separated, as previously mentioned. A disadvantage is that the template mechanism becomes too rigid compared to the flexibility of the `print`-like statements available in the script-centered languages. However, those languages permit essentially no static guarantees or work separation. Furthermore, with the script-centered solutions the HTML must often be constructed in a linear fashion from top to bottom, instead of being composed from components in a more logical manner. The `<bigwig>` solution provides the best of the two worlds. Higher-order HTML templates as first-class values are in practice as flexible as `print` statements, and the MAWL benefits are still preserved.

We define *DynDoc* as the sub-language of `<bigwig>` that deals with document construction, that is, the control structures, HTML template constants, variables and assignments, plug operations, and `show-receive` statements. Template constants are delimited by `<html>...</html>`. Gaps are written with special `<[...]>` tags. Special *attribute gaps* can be used in place of attribute values, as shown in the example below. Of course, only strings should be plugged into such gaps, not templates with markup. The plug operation `x<[y=z]>` creates a new template by inserting a copy of `z` in the `y` gaps of a copy of `x`. When used in a `show-receive` statement, a template is converted to a complete document by implicitly plugging empty strings into all remaining gaps. Also, it is automatically wrapped into a `form` element whose action is to continue the session, unless the session terminates immediately after. And finally, it is inserted into an outermost template like

```
<html><head><title>service</title></head><body>...</body></html>
```

unless already inside a body element. The following example illustrates that documents can be built gradually using higher-order templates:

```
service {
  html brics = <html>
    <head><title>Hi!</title></head>
    <body bgcolor=[color]><[contents]></body>
```

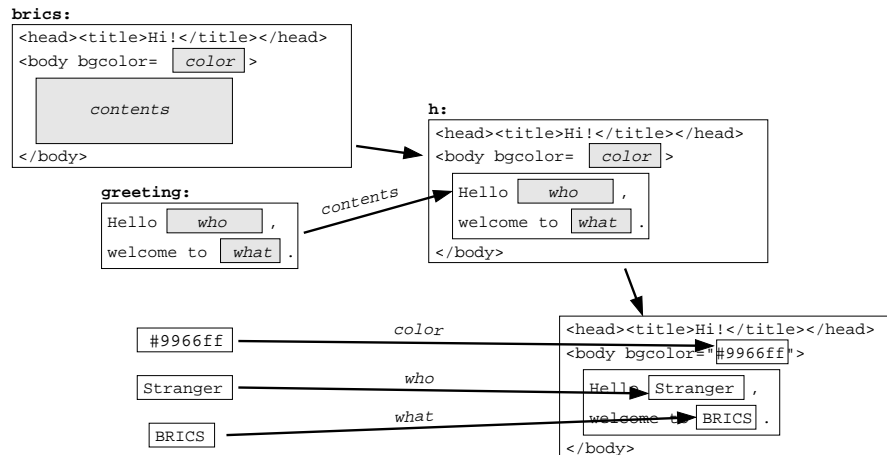


Figure 9.3: Building a document by plugging into template gaps. The construction starts with the five constants on the left and ends with the complete document on the right.

```

</html>;
html greeting = <html>Hello <[who]>, welcome to <[what]>.</html>;
session Welcome() {
  html h = brics<[contents=greeting]>;
  show h<[color="#9966ff",who="Stranger",what="BRICS"]>;
}

```

The construction process is shown in Figure 9.3. Note that gaps may be plugged in any order, not necessarily “bottom up”. MAWL provides little functionality beyond plugging text strings into gaps. The special MITER tag allows list structures to be built iteratively, but still precludes general tree-like structures. The following <bigwig> example uses a recursive function to construct an HTML document representing a binary tree:

```

service {
  html list = <html><ul><li><[gap]><li><[gap]></li></ul></html>;
  html tree(int i) {
    if (i==0) return <html>foo</html>;
    return list<[gap=tree(i-1)]>;
  }
  session ShowTree() {
    show tree(10);
  }
}

```

Something similar could not be done with MAWL’s first-order templates. In a script-centered or a page-centered language it is of course possible, but not with such a simple program structure reflecting the logical composition of the document, since it must be generated linearly by printing to the output stream. An alternative is to use an HTML tree constructor library, but that forces documents to be built bottom-up, which is often inconvenient.

The use of higher-order templates generally leads to programs with a large number of relatively small template constants. For that reason it is convenient to be able to

inline the constants in the program code, as in these examples, rather than always placing them in separate files. However, we do offer explicit support for factoring out the work of graphical designers using a `#include` construct as in C. Alternatively, each HTML constant in a `<bigwig>` program may have an associated URL, pointing to an alternate, presumably more elaborate, version:

```
service {
  session Hello {
    show <html>Hello World</html> @ "fancy/hello.html";
  }
}
```

The compiler retrieves the indicated file and uses its contents in place of the constant, provided it exists and contains well-formed HTML. In this manner, the programmer can use plain versions of the templates while a graphical designer simultaneously produces fancy versions. The compiler checks that the two versions have the same gaps and fields. In order to accommodate the use of HTML authoring tools, we permit gaps to be specified in an alternative syntax using special tags.

The DynDoc sub-language was introduced in [195] where it is also shown how this template model can be implemented efficiently with a compact runtime representation. The plug operation takes only constant time, and showing a document takes time linear in the size of the output. Also, the size of the runtime representation of a document may be only a fraction of its printed size. For example, a binary tree of height n shown earlier has a representation of size $O(n)$ rather than $O(2^n)$.

9.3.1 Analysis of Template Construction and Form Input

We wish to devise a type checker that allows as liberal a use of dynamic documents as possible, while guaranteeing that no errors occur. More precisely, we would like to verify the following properties at compile time:

- at every plug operation, $x < [y=z]$, there always exists a y gap in x ;
- the gap types are compatible with the values being plugged in, in particular, HTML with markup tags is never inserted into attribute gaps;
- for every `show-receive` statement, the fields in the `receive` part always exist in the document being shown;
- the field types are compatible with the `receive` parts, for instance, a select menu allowing multiple items to be selected yields a vector value; and
- only valid HTML 4.01 [185] is ever sent to the clients.

The first four properties are addressed in [195] as summarized below. The last property is covered in the following section.

It is infeasible to explicitly declare the exact types of higher-order templates for two reasons. First, all gaps and all fields and their individual capabilities would have to be described, which may become rather voluminous. Second, this would also imply that an HTML variable has the same type at every program point, which is too restrictive to allow templates to be composed in an intuitive manner. Consequently, we rely

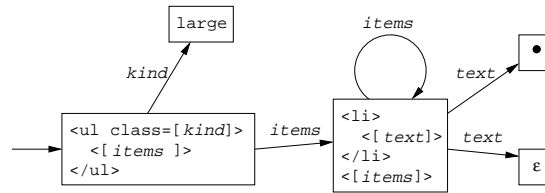


Figure 9.4: A summary graph representing a set of HTML fragments.

instead on a flow analysis to infer the types of template variables and expressions at every program point. In our experience, this results in a liberal and useful mechanism.

We employ a monovariant interprocedural flow analysis, which guarantees that the form fields in a shown document correspond to those that are received, and that gaps are always present when they are being plugged. This analysis fits into standard data-flow frameworks [174], however it applies a highly specialized lattice structure representing the template types. For every template variable and expression that occurs in the given program, we associate a lattice element that abstractly captures the relevant template properties. The lattice consists of two components: a *gap map* and a *field map*. The gap map records for every occurring gap name whether or not the gap occurs at that point, and in case it does occur, whether it is an HTML gap or an attribute gap. Similarly, the field map records for every occurring input field name information about the input fields, which can be of type text, radio, select, or checkbox, representing the different interaction methods.

Given a <bigwig> program we construct a flow graph. This is quite easy since there are no higher-order functions or virtual methods. All language constructs that are not included in DynDoc are abstracted away. It is now possible to define transfer functions that abstractly describe the effect of the program statements. This produces a constraint system which we solve using a classical fixed point iteration technique. From this solution, we can see that the first three properties mentioned above are satisfied, and, if not, generate error messages indicating the cause.

With this approach, the programmer is only restricted by the requirement that at every program point the template type of an expression must be fixed. In practice, this does not limit the expressibility, rather, it tends to enforce a more comprehensible structure of the programs. Also, the compiler silently resolves conflicts at flow join points by implicitly plugging superfluous gaps with empty content.

9.3.2 HTML Validity Analysis

The fifth property, HTML validity, is addressed with a similar but more complicated approach, as described in [39].

The main idea is the following: We define a finite structure called a *summary graph* that approximates the set of templates that a given HTML expression may evaluate to. This structure contains the plug operations and the constant templates and strings that are involved.

As an example, consider the summary graph in Figure 9.4. The nodes correspond to program constants and the edges correspond to plug operations. For instance, the `li` template may be plugged into the `items` gaps in the `ul` template here. The node

labeled \bullet represents arbitrary text strings and ϵ is the empty string. The root of the graph corresponds to the outermost template. By “unfolding” this graph according to the plug edges, this summary graph defines a possibly infinite set of HTML fragments without gaps (in this case the set of all `ul` lists of class `large` with one or more character data items). This structure turns out to provide an ideal abstraction level for verifying HTML validity.

Again, we apply a data-flow analysis to approximate the flow of template values in the program. This time we use a lattice consisting of summary graphs. It is possible to model plug operations with good precision using transfer functions; however, two preliminary analyses are required: one for tracking string constants, and one, called a *gap track analysis*, for tracking the origins of gaps. The latter tells us, for each template variable and gap name, which constant templates containing such a gap can flow into that variable at any given program point. Clearly, these analyses are highly specialized for the domain of dynamic document construction and for `<bigwig>`’s higher-order template mechanism, but they all fit into the standard data-flow analysis frameworks. For more details see [39].

Once we have the summary graphs for all the `show` statements, we need to verify that the sets of document fragments they define are all valid HTML according to W3C’s official definition. To simplify the process we reformulate the notion of Document Type Definition (DTD) as a simpler and more convenient formalism which we call *abstract DTD*. An abstract DTD consists of a number of *element declarations*, whereof one is designated as the root. An element declaration defines the requirements for a particular type of element. Each declaration consists of an element name, a set of names of attributes and subelements that may occur, and a boolean expression constraining the element type instances with respect to their attribute values and contents. The official DTD for HTML is easily rewritten into our abstract DTD notation. In fact, the abstract DTD version captures more validity requirements than those expressible by standard DTDs and merely appear as comments in the HTML DTD. As a technicality, we actually work with XHTML 1.0 which is an XML reformulation of HTML 4.01. There are no conceptual differences, except that the XML version provides a cleaner tree view of documents for the analysis.

Given a summary graph and an abstract DTD description of HTML, validity can be checked by a recursive traversal of the summary graph starting at the roots. We memoize intermediate results to ensure termination, since the summary graphs may contain loops. If no violations are encountered, the summary graph is valid. Since all validity properties are local to single elements and their contents, we are able to produce precise error messages in case of violations. Analysis soundness is ensured by the following property: if all summary graphs corresponding to `show` expressions are verified to be valid with respect to the abstract DTD, then all concrete documents are guaranteed to be valid HTML.

The program analyses described here all have high worst-case complexities due to complex lattices. Nevertheless, our implementations and experiments show that they work well in practice, even for large intricate programs. These experiments are mentioned in Section 9.8.

9.3.3 Caching of Dynamically Generated HTML

Traditional Web caching based on HTTP works by associating an expiration time to all documents sent by the servers to the clients. This has helped in decreasing both network and server load and response times. By default, no expiration is set, and, by using “now”, caching is effectively disabled. This technique was designed primarily for documents whose contents rarely or never change, not for documents dynamically generated by interactive Web services. The gradual change from statically to dynamically generated documents has therefore caused the impact of Web caching to degrade.

Existing proposals addressing this include Active Cache, HPP, and various server-based techniques, as explained in the survey in [36]. Server-based techniques aim to relieve the server of redundant computations, not to decrease network load. They typically work by simplifying assumptions, for instance that many interactions can be handled without side-effects on the global service state, that interactions are often identical for many clients, or that the dynamics of the pages is limited to, e.g., banner ad rotation. None of this applies to complex interactive services. Active Cache is a proxy-based solution that employs programmable cache applets. This can be very effective, but it requires both specialized proxy servers and careful programming to ensure consistency between the proxies and the main server.

HPP tries to separate the constant parts from the dynamic parts of the generated documents. We apply a similar technique. In contrast to HPP, our solution is entirely automatic, while HPP requires extra programming. The idea is to exploit the clear division between the service code and the HTML templates in <bigwig>. In our normal implementation of DynDoc, the internal template representation is converted to an HTML document on the server when the `show` statement is executed. Instead, we now store each template constant in a fixed file on the server, and defer the conversion to the client using a JavaScript representation of the dynamic parts. The template files can now be cached by the ordinary browser caches. More details of the technique can be found in [36]. We summarize our evaluation results in Section 9.8.

9.3.4 Code Gaps and Document Clusters

In the following, we describe two extensions to the DynDoc language. Occasionally, the page-centered approach is admittedly more appropriate than the session-centered one. Consider the following example, which gives the current time of day:

```
service {
  session Time() {
    html h = <html>Right now, the time is <[t]></html>;
    show h<[t=now()]>;
  }
}
```

An equivalent but less clumsy version can be written using *code gaps*, which implicitly represent expressions whose values are computed and plugged into gaps when the document is being shown:

```
service {
  session Time() {
    html h = <html>Right now, the time is <[(now())]></html>;
  }
}
```

```

    show h;
  }
}

```

Documents with code gaps remain first-class values, since the code can only access the global scope. Note that code gaps in `<bigwig>` are more powerful than the usual page-centered approach, since the code exists in the full context of sessions, shared variables, and concurrency control. In fact, with the idea of *published* documents described in Section 9.6, the page-centered approach is now included as a special case of `<bigwig>`.

Some services may want to offer the client more than a single document to browse, for example, the response could be a tiny customized Web site. In `<bigwig>` we have experimented with support for showing such *document clusters*. The difficulty is to provide a simple notation for specifying an arbitrary graph of documents connected by links. For an HTML variable `x`, we introduce the *document reference* notation `&x`, which can be used as the right-hand side of a plug operation. It will eventually expand into a URL, but not until the document is finally shown. Until then, the flow analysis just records the connection between the gap and the variable. When a document is shown, the transitive closure of document references is computed, and the resulting cluster of documents is produced with references replaced by corresponding URLs. The following example shows a cluster of two documents that are cyclically connected. Notice that the cluster can be browsed freely without cluttering the control-flow:

```

service {
  session Cluster() {
    html greeting = <html>
      Hi! Click <a href=[where]>here</a> for a kind word.
    </html>;
    html kind = <html>How nice to see you! <a href=[there]>Back</a></html>;
    kind = kind<[there = &Greeting];
    show greeting<[where=&kind];
  }
}

```

The compiler checks that all cluster documents with submit buttons contain the same form fields. It is also necessary to perform an escape analysis to ensure that document variables are not exported out of their scope.

9.4 Form Field Validation

A considerable effort in Web programming is expended on form field validation, that is, on checking whether the data supplied by the client in the form fields is valid, and when it is not, producing error messages and requesting the fields to be filled in again. Apart from details about regular expression matching, the main problem is to program a solution that is robust, efficient, and user friendly.

One approach is *server-side* validation, where the form fields are validated on the server when the page has been submitted. None of the languages mentioned in Section 9.1 provide any help for this, except for the regular expression matching in Perl. Therefore, the main logic of the service often becomes cluttered with validation code.

In a sense, every program part that sends a page to a client must be wrapped into a while-loop that repeats until the input is valid. Other disadvantages include wasting bandwidth and causing delays to the users.

The alternative is *client-side* validation, which usually requires the programmer to use JavaScript in the pages being generated. This permits more sophisticated user interactions and reduces the communication overhead. However, client-side validation should not be used alone. The reason is that the client is perfectly capable of bypassing the JavaScript code, so an additional server side validation must always be performed. Thus, the same code must essentially be written both in JavaScript and in the server scripting language. In practice, writing JavaScript input validators that capture all validity requirements and at the same time are also user friendly can be very difficult, since, unfortunately, most browsers differ in their JavaScript support. Whole Web sites are dedicated to explaining how the various subsets of JavaScript work in different browsers.³

In <bigwig> we have introduced a domain-specific sub-language, called PowerForms, for form field validation [37]. It handles complex interdependencies between form fields, and the compiler generates the required code for both client and server. By compiling into JavaScript, only the PowerForms implementors need to know the details of how browsers support JavaScript, rather than all Web service programmers. Also, the programmer does not need to write essentially the same code in a server-side version and a client-side version anymore.

PowerForms is a declarative language. Informally, this means that the programmer specifies what the input requirements are, not how to check them. In its simplest form, PowerForms allows regular-expression *formats* to be associated to form fields:

```
service {
  format Digit = range('0','9');
  format Alpha = union(range('a','z'),range('A','Z'));
  format Word = concat(Alpha,star(union(Digit,Alpha)));
  format Email = concat(Word,"@",Word,star(concat(".",Word)));
  session Validate() {
    html form = <html>
      Please enter your email address:
      <input name=email type=text size=20>
      <format name=Email field=email>
    </html>;
    string s;
    show form receive[s=email];
  }
}
```

This example shows how to constrain input in the email field to a certain regular expression. The <bigwig> compiler generates the JavaScript code that checks the user input on the client-side and provides help and error messages, and also the code that performs the server-side double-check. “Traffic-light” icons next to the input fields provide the user with continuous feedback about the string entered so far. “Green” means valid, “yellow” means invalid but a prefix of something valid, and “red” means not a prefix of something valid. Other alternatives can be chosen, such as checkmark

³See e.g. <http://www.webdevelopersjournal.com/articles/javascript-limitations.html> or <http://www.xs4all.nl/~ppk/js/version5.html>.

symbols, arrows, etc. We also allow the usual UNIX-style syntax for regular expressions in the subset of our notation that excludes the intersection and complement operators.

Formats can be associated to all kinds of form fields, not just those of type `text`. For `select` fields, the format is used to filter the available options. For `radio` and `checkbox` fields, only the permitted buttons can be depressed.

As noted in [79], many forms contain fields whose values may be constrained by those entered in other fields. A typical example is a field that is not applicable if some other field has a certain value. Such interdependencies are almost always handled on the server, even if the rest of the validation is performed on the client. Presumably, the reason is that interdependencies require even more delicate JavaScript programming. The `<bigwig>` solution is to allow such field interdependencies to be specified using an extension of the regular expressions: the `format` tags are extended to describe boolean decision trees, whose conditions probe the values of other form fields and whose leaves are simple formats. The interdependence is resolved by a fixed-point process computed on the client by JavaScript code automatically generated by the `<bigwig>` compiler. A simple example is the following, where the client chooses a letter group and the `select` menu is then dynamically restricted to those letters:

```
service {
  format Vowel = charset("aeiouy");
  format Consonant = charset("bcdfghjklmnpqrstvwzz");
  html form = <html>
    Favorite letter group:
    <input type=radio name=group value=vowel checked>vowels
    <input type=radio name=group value=consonant>consonants
    <br>
    Favorite letter:
    <select name=letter>
      <option value="a">a
      <option value="b">b
      <option value="c">c
      ...
      <option value="z">z
    </select>
    <format field=letter>
      <if><equal field=group value=vowel>
        <then><format name=Vowel></then>
        <else><format name=Consonant></else>
      </if>
    </format>
  </html>;
  session Letter() {
    string s;
    show form receive[s=letter];
  }
}
```

ColdFusion [46] provides a mechanism reminiscent of PowerForms. However, it does not support field interdependencies or validation of non-text fields. PowerForms is shown to be a simple language with a clean semantics that appears to handle most realistic situations. We have implemented it both as part of the `<bigwig>` compiler

and as a stand-alone tool that can be used to add input validation to general HTML pages.

9.5 Concurrency Control

As services have several session threads, there is a need for synchronization and other concurrency controls to discipline the concurrent behavior of the active threads. A simple case is to control access to the shared variables using mutex regions or the readers/writers protocol. Another issue is enforcement of priorities between different session kinds, such that a management session may block other sessions from running. Another example is event handling, where a session thread may wait for certain events to be caused by other threads.

We deal with all of these scenarios in a uniform manner, based on a central controller process in the runtime system, which is general enough to enforce a wide range of safety properties [194]. The support for concurrency control in the previously mentioned Web languages is limited to more traditional solutions, such as file locking, monitor regions, or synchronized methods.

A <bigwig> service has an associated set of *event labels*. During execution, a session thread may request permission from the controller to pass a specific event checkpoint. Until such permission is granted, the session thread is suspended. The policy of the controller must be programmed to maintain the appropriate global invariants for the entire service. Clearly, this calls for a domain-specific sub-language. We have chosen a well-known and very general formalism, temporal logic. In particular, we use a variation of monadic second-order logic [208]. A formula describes a set of strings of event labels, and the associated semantics is that the trace of all event labels being passed by all threads must belong to that set. To guide the controller, the <bigwig> compiler uses the MONA tool [131] to translate the given formula into a minimal deterministic finite-state automaton that is used by the controller process to grant permissions to individual threads. When a thread asks to pass a given event label, it is placed in a corresponding queue. The controller continually looks for nonempty queues whose event labels correspond to enabled transitions from the current DFA state. When a match is found, the corresponding transition is performed and the chosen thread is resumed. Of course, the controller must be implemented to satisfy some fairness requirements. All regular trace languages can be expressed in the logic.

Applying temporal logics is a very abstract approach that can be harsh on the average programmer. However, using syntax macros, which are described in Section 9.6, it is possible to capture common concurrency primitives, such as semaphores, mutex regions, the readers/writers protocol, monitors, and so on, and provide high-level language constructs hiding the actual formulas. The advantage is that <bigwig> can be extended with any such constructs, even some that are highly customized to particular applications, while maintaining a simple core language for concurrency control.

The following example illustrates a simple service that implements a critical region using the event labels `enter` and `leave`:

```
service {  
  shared int i;  
  session Critical() {
```

```

constraint {
    label leave,enter;
    all t1,t3: (t1<t3 && enter(t1) && enter(t3)) =>
        is t2: t1<t2 && t2<t3 && leave(t2);
}
wait enter;
i = i+1;
wait leave;
}

```

The formula states that for any two enter events there is a leave event in between, which implies that at any time at most one thread is allowed in the critical region. Using syntax macros, programmers are allowed to build higher-level abstractions such that the following can be written instead:

```

service {
    shared int i;
    session Critical() {
        region {
            i = i+1;
        }
    }
}

```

We omit the macro definitions here. In its full generality, the wait statement is more like a switch statement that allows a thread to simultaneously attempt to pass several event labels and request a timeout after waiting a specified time.

A different example implements an asynchronous event handler. Without the macros, this could be programmed as

```

service {
    shared int i;
    constraint {
        label handle,cause;
        all t1: handle(t1) => is t2: t2<t1 && cause(t2) &&
            (all t3: t2<t3 && t3<t1 => !handle(t3));
    }
    session Handler() {
        while (true) {
            wait handle;
            i++;
        }
    }
    session Application() {
        wait cause;
    }
}

```

This nontrivial formula allows the handler to proceed, without blocking the application, whenever the associated event has been caused at least once since the last invocation of the handler. Fortunately, the macros again permit high-level abstractions to be introduced with more palatable syntax:

```

service {
  shared int i;
  event Increment {
    i++;
  }
  session Application() {
    cause Increment;
  }
}

```

The runtime model with a centralized controller process that ensures satisfaction of safety constraints is described in [38]. The use of monadic second-order logic for controller synthesis was introduced in [194] where additionally the notions of *triggers* and *counters* are introduced to gain expressive power beyond regular sets of traces, and conditions for distributing the controller for better performance are defined.

The session model provides an opportunity to get a global view of the concurrent behavior of a service. Our current approach does not exploit this knowledge of the control flow. However, we plan to investigate how it can be used in specialized program analyses that check whether liveness and other concurrency requirements are complied with.

9.6 Syntax Macros

As previously mentioned, <bigwig> contains a notion of macros. Although not specific to Web services, this abstraction mechanism is an essential part of <bigwig> that serves to keep the sub-languages minimal and to tie them together.

A macro language can be characterized by its level of operation which is either *lexical* or *syntactic*. Lexical macro languages operate on sequences of tokens and conceptually precede parsing. Due to the independence of syntax, macros often have unintended effects, and parse errors are only discovered at invocation time. Consequently, programmers are required to consider how individual macro invocations are being expanded and parsed. Syntactic macros amend this by operating on parse trees instead of token sequences [219]. Types are added to the macro arguments and bodies in the form of nonterminals of the host language grammar. Macro definitions can now be syntax-checked at definition time, guaranteeing that parse errors no longer occur as a consequence of macro expansion. Using syntax macros, the syntax of the programming language simply appears to be extended with new productions.

Our macros are syntactic and based entirely on simple declarative concepts such as grammars and substitution, making them easy and safe to use by ordinary Web service programmers. Other macro languages, such as MS², Scheme macros, and Maya, instead apply full Turing complete programming languages for manipulating parse trees at compile time, making them more difficult to use.

As an initial example, we extend the core language of <bigwig> with a `repeat-until` control structure that is easily defined in terms of a `while` loop.

```

macro <stm> repeat <stm S> until ( <exp E> ) ; ::= {
{
  bool first = true;
  while (first || !<E>) {

```



```

    <S>
    first = false;
  }
}
}

```

The first line is the header of the macro definition. It specifies the nonterminal type of the macro abstraction and the invocation syntax, including the typed arguments. As expected, the type of the `repeat-until` macro is `<stm>`, representing statements. This causes the body of the macro to be parsed as a statement and announces that invocations are only allowed in places where an ordinary statement would be. We allow the programmer to design the invocation syntax of the macro. This is used to guide parsing and adds to the transparency of the macro abstractions. This particular macro is designed to parse two appropriately delimited arguments, a statement `S` and an expression `E`. The body of the macro implements the abstraction using a boolean variable and a `while` loop. When the macro is invoked, the identifiers occurring in the body are α -converted to avoid name clashes with the invocation context.

With a concept of *packages*, macros can be bundled up in collections. Our experience with `<bigwig>` programming has led us to develop a “standard macro package”, `std.wigmac`, that extends the sub-languages of `<bigwig>` in various ways and has helped keep the language minimal. For instance, the form field validation language is extended with an optional regular expression construct, and database language macros transform SQL-like queries into our own iterative `factor` construction. Also, various composite security modifiers are defined, and concurrency control macros, such as the `region` from Section 9.5, gradually build on top of each other to implement increasingly sophisticated abstractions.

Macros are also used to tie together different sub-languages, making them collaborate to provide tailor-made extensions of the language. For instance, the sub-languages dealing with sessions, dynamic documents, and concurrency control can be combined into a `publish` macro. This macro is useful when a service wishes to publish a page that is mostly static, yet needs to be recomputed once in a while, when the underlying data changes. The following macros efficiently implements such an abstraction:

```

macro <toplevels> publish <id D> { <exp E> } ::= {
  shared html <D>~cache;
  shared bool <D>~cached;
  session <D>() {
    exclusive if (!<D>~cached) {
      <D>~cache = <E>;
      <D>~cached = true;
    }
    show <D>~cache;
  }
}

macro <stm> touch <id d> ; ::= {
  <d>~cached = false;
}

```

The `publish` macro recomputes the document if the cache has expired and then shows the document, while the `touch` macro causes the cache to expire. The `~` operator is

used to create new identifiers by concatenating others. Using this extended syntax, a service maintaining a high-score list, for example, can look like this:

```
require "publish.wigmac"
service {
  shared int record;
  shared string holder;
  publish HiScore {
    computeWinnerDoc(record, holder)
  }
  session Play() {
    int score = play();
    if (score>=record) {
      show EnterName receive[holder=name];
      record = score;
      touch HiScore;
    } else {
      show <html>Sorry, no record.</html>;
    }
  }
}
```

Here the high-score document is only regenerated when a player beats the record. This code is clearly easier to understand and maintain than the corresponding expanded code.

The expressive power of syntax macros is extended with a concept of *metamorphisms*, as explained in [41]. This declaratively permits tree structures to be transformed into host language syntax without compromising syntactic safety, something not possible with other macro languages. Using this mechanism in an extreme way, it is possible to define whole new languages. We call this concept a *very* domain-specific language, or VDSL.

At the University of Aarhus, undergraduate computer science students must complete a bachelor's degree in one of several fields. The requirements that must be satisfied are surprisingly complicated. To guide them towards this goal, the students must maintain a so-called "bachelor's contract" that plans their remaining studies and discovers potential problems. This process is supported by a Web service, which, for each student, iteratively accepts past and future course activities, checks them against all requirements, and diagnoses violations until a legal contract is composed. This service was first written as a straight <bigwig> application, but quickly became annoying to maintain due to constant changes in the curriculum. It was redesigned in the form of a VDSL, where study fields and requirements are conceptualized and defined directly in a more natural language style. This makes it possible for non-programmers to maintain and update the service. An small example input is

```
require "bachelor.wigmac"
studies
  course Math101
    title "Mathematics 101"
    2 points fall term
  ...
  course Phys202
    title "Physics 202"
```

```

    2 points spring term
course Lab304
  title "Lab Work 304"
  1 point fall term
exclusions
  Math101 <> MathA
  Math102 <> MathB
prerequisites
  Math101,Math102 < Math201,Math202,Math203,Math204
  CS101,CS102 < CS201,CS203
  Math101,CS101 < CS202
  Math101 < Stat101
  CS202,CS203 < CS301,CS302,CS303,CS304
  Phys101,Phys102 < Phys201,Phys202,Phys203,Phys301
  Phys203 < Phys302,Phys303,Lab301,Lab302,Lab303
  Lab101,Lab102 < Lab201,Lab202
  Lab201,Lab202 < Lab301,Lab302,Lab303,Lab304
field "CS-Mathematics"
  field courses
    Math101,Math102,Math201,Math202,Stat101,CS101,CS102,CS201,CS202,CS203,
    CS204,CS301,CS302,CS303, CS304,Project
  other courses
    MathA,MathB,Math203,Math204,Phys101,Phys102,Phys201,Phys202
constraints
  has passed CS101,CS102
  at least 2 courses among CS201,CS202,CS203
  at least one of Math201,Math202
  at least 2 courses among Stat101,Math202,Math203
  has 4 points among Project,CS303,CS304
  in total between 36 and 40 points

```

None of the syntax displayed is plain <bigwig>, except the macro package require instruction. The entire program is the argument to a single macro, `studies`, that expands into the complete code for a corresponding Web service. The file `bachelor.wigmac` is only 400 lines and yet defines a complete implementation of the new language. Thus, the <bigwig> macro mechanism offers a rapid and inexpensive realization of new ad-hoc languages with almost any syntax desired. Similar features do not occur in any of the Web service languages mentioned in the previous sections.

9.7 Other Web Service Aspects

There are of course other features in <bigwig> that are necessary to support Web service development, but for which we have no major innovations. These are briefly presented in this section.

9.7.1 HTML Deconstruction

The template mechanism is used to construct HTML documents, but when “run in reverse” it also allows for deconstruction. This is realized by using the templates as patterns in which the gaps play the role of variables, as illustrated in this example:

```

service {
  html Template = <html>

```

```

    <[]><img src=[source] alt="today's Dilbert comic"><[]>
</html>;
session Dilbert() {
    string data = get("http://www.dilbert.com/");
    string s;
    match(data, Template) [s=source];
    exit Template<[source="http://www.dilbert.com"+s];
}
}

```

which grabs the daily strip from the Dilbert home page. Gaps without names serve as wildcards.

9.7.2 Seslets

For some interaction patterns a strict session model can be inappropriate, since the client and server must alternate between being active and suspended. Furthermore, information cannot be pushed on the server's initiative while the client is viewing a page. A simple example is a chat room where new messages should appear automatically, without the client having to reload the page being viewed, and where only the new message and not the entire new page is transmitted. The essence of this concept is *client-side computations*, which are able to contact the server on their own accord.

The <bigwig> solution is a notion of *seslets*. A seslet is a kind of lightweight session that is allowed to do anything an ordinary session can do, except perform *show* operations. It is invoked by the client with some arguments and eventually returns a reply of any <bigwig> type. Typically, it performs database operations or waits for certain events to occur, and then reports back to the client.

Since we are limited by the existing technologies on the client-side, our current implementation is restricted to using Java applets or JavaScript. To facilitate the writing of applets, the <bigwig> compiler generates the Java code for an abstract class extending *Applet*, which must be inherited from in order to access the available seslets. Alternatively, we have experimented with a JavaScript interface. However, this approach is limited by the lack of client-server communication support from JavaScript, so we currently apply cookies for the communication.

An important use of seslets is to allow client-side code to synchronize with other active threads on the server. For example, the chat room solution could employ a seslet that uses the concurrency control mechanisms of <bigwig> to wait until the next message is available, which is then returned to the applet. In this way, no client pulling or busy waiting is required.

9.7.3 Databases

Most Web services are centered around a database. In the general case, this is an existing, external database which the service must connect to. The <bigwig> system supports the ODBC interface for this purpose. In most other Web service languages, database queries are built dynamically as strings that must be parsed by the database engine. In <bigwig>, queries are not built as strings but are written in a query language that is part of the <bigwig> syntax. This allows for compile-time checking of the syntax and types of queries, eliminating another source of errors. Since many smaller

services use only simple data, we also offer an internal database that is implemented on top of the file system.

9.7.4 Security

There are many aspects of Web service security.⁴ The security in <bigwig> can be divided into two categories, depending on whether it is generically applicable to all services or specific to the behavior of a particular service.

The former category mostly relates to the runtime environment and communication, dealing with concepts such as integrity, authenticity, and confidentiality. Integrity of a session thread's local state is achieved by keeping it exclusively on the server. Integrity of shared data is provided by the database. An interaction key is generated and embedded in every document shown to the client and serves to prevent submission of old documents. Clients and session threads are associated through a random key which is created by the server upon the first request and carried across interactions in a hidden input field. This mechanism may optionally be combined with other security measures, such as SSL, to provide the necessary level of security. Authenticity and confidentiality are addressed through general declarative security modifiers that the programmer can attach on a *service*, *session*, or *show* basis. The modifiers *ssl* and *htaccess* enforce that the SSL and HTTP Authentication protocols are used for communication. The *selective* modifier restricts access to a session to those clients whose IP numbers match a given set of prefixes. Finally, the *singular* modifier ensures that the client has the same IP address throughout the execution of a session.

We envision performing some simple static analyses relating to the behavioral security of particular services. Values are classified as *secret* or *trusted*, and, in contrast to tainting in Perl, the compiler keeps track of the propagation of these properties. Furthermore, there are restrictions on how each kind of data can be used. Form data is always assumed to be untrusted and gaps are never allowed to be plugged with secret values. Variables can be declared with the modifiers *secret* or *trusted* and may then only contain the corresponding values. The *system* function can only be called with a trusted string value. To change the classification of a value, there are two functions, *trust* and *disclose*. The programmer must make the explicit choice of using these coercions. An example involving trust is the following service:

```
service {
  session Lookup() {
    html Error = <html>Invalid URL!</html>;
    html EnterURL = <html>Enter a URL: <input type=text name=URL></html>;
    string u, domain;
    show EnterURL receive[u = URL];
    if (|u|<7 || u[0..7]!="http://") show Error;
    for (i=7; i<|u| && u[i]!='/' ; i++);
    domain = u[7..i];
    if (system("/usr/sbin/nslookup '" + domain + "'").stderr!="") {
      show Error;
    }
  }
}
```

⁴See <http://www.w3.org/Security/faq/>.

This code performs an *nslookup* on the URL supplied by the user to check whether its domain exists. Since the value of `domain` is derived from the form field `URL` it should not be trusted, and its use in the call of `system` will be flagged by the compiler. And, indeed, it would be unfortunate if the client enters `"http://foo';rm -rf /'"` in the form. A similar analysis is performed for `secret`. Consider the example:

```
service {
  shared secret string password;
  bool odd(int n) { return n%2==1; }
  session Reveal() {
    if (odd(|password|)) show <html>foo</html>;
  }
}
```

The compiler is sufficiently paranoid to reject this program, since the branching of the `if`-statement depends on a function applied to information derived from a secret value. These analyses are not particularly original, but are not seen in other Web service programming languages.

There is still much work to be done in this area. So far, we have not considered using cryptological techniques to ensure service integrity, the role of certificates, or more sophisticated static analyses.

9.8 Evaluation

The <bigwig> language should be evaluated according to two different criteria. First, the quality of our language design as seen from concrete programming experiences. This is necessarily a somewhat intangible and subjective criterion. Second, the performance of our language implementation as seen from observed benchmarks.

9.8.1 Experience with <bigwig>

<bigwig> is still mainly an experimental research tool, but we have gained experiences from numerous minor services that we have written for our own edification, a good number of services that are used for administrative purposes at the University of Aarhus, and a couple of production services on which we have collaborated. Apart from these applications, we estimate that <bigwig> has been downloaded roughly 2500 times from our Web site, and we have mainly received positive feedback from the users.

One production service is the Web site of the European Association for Theoretical Computer Science (www.eatcs.org), handling newsletters, webboards, and several membership services. It is written in 5,345 lines of <bigwig>, using 133 HTML templates and 114 `show` statements. Another is the Web site of the JAOO 2001 conference (www.jaoo.dk), handling all aspects of advertisement, schedules, registration, and attendance services. It is written in 7,943 lines of <bigwig>, using 248 HTML templates, and 39 `show` statements.

These experiences have shown that <bigwig> has two very strong points. First, the session concept greatly simplifies the programming of complicated control flow with multiple client interactions. Second, the HTML templates are very easy and intuitive

to use and the static guarantees catching numerous errors, many of which are difficult to find by any other means. It is particularly helpful that the HTML analyzers provide precise and intuitive error messages.

The JAOO application has been particularly interesting, since it involves collaboration with an external HTML designer. This experience confirmed that our templates are successful in defining an interface between programmers and designers and that gaps and fields define a useful contract between the two.

The main weak point that we identified is the core language, which is often found to lack minor features. We plan to address this in future work, as mentioned in Section 9.9.

The stand-alone version of the PowerForms sub-language has been surprisingly popular in its own right. It has many active users, and has been integrated into a proprietary Web deployment system.

9.8.2 Performance

When evaluating the performance of the `<bigwig>` implementation, we want to focus on the areas where we tried to provide improvements. We are not aiming for simple high-load services, but are focusing on services with intricate control-flow. Still, informal tests show that the throughput of our services is certainly comparable with that of straight CGI-based services or Servlet applications running on J2SE.

The automatic caching scheme based on our HTML templates is designed to exploit their intricate structure to cache static fragments on the client side. We have obtained real benefits from this approach. The experiments reported in [36] show that the size of the transmitted data may shrink by a factor of 3 to 30, which on a dial-up connection translates into a reduction in download time by a factor of 2 to 10.

It is also relevant to evaluate the performance of the `<bigwig>` compiler, since we employ a series of theoretically quite expensive static analyses. However, in practice they perform very well, as documented in [195, 39]. The EATCS service is analyzed for HTML validity in 6.7 seconds and the JAOO service in 2.4 seconds.

9.9 Conclusion

The `<bigwig>` project has identified central aspects of interactive Web services and provided solutions in a coherent framework based on programming language theory. At the same time, the `<bigwig>` project is a case study in applications of the domain-specific language design paradigm.

We argue that the notion of sessions is essential to Web services and should constitute the basic structure of a Web service programming language. Together with higher-order document templates, such as in the DynDoc sub-language, the dynamic construction of Web pages becomes flexible at the same time, making it easy to use, and safe by compile-time guarantees regarding document validity and the use of input forms. We have shown that form field validation, compared to traditional approaches, can be made easier with a domain-specific sub-language, such as PowerForms, which automatically translates high-level specifications into a combination of more low-level server-side and client-side code. We have examined how temporal logics can be used to

synthesize concurrency controllers. Finally, we have demonstrated how macro mechanisms can be made effective for extending and combining languages, in the context of the sub-languages of <bigwig>.

Version 2.0 of the <bigwig> compiler and runtime system is freely available from the project home page at www.brics.dk/bigwig/ where documentation and examples can also be found.

Regarding the future development of <bigwig> we now move towards Java. We are developing **JWIG** [57] as an extension of Java, where we add the most successful features of <bigwig>, such as the session model, dynamic documents, form field validation, and syntax macros. Since the design of <bigwig> has focused on the Web specific areas, we hope that the many standard programming issues of Web services become easier to develop with **JWIG**. However, a number of new challenges arise. For instance, the program analyses described in Section 9.3 all assume that we have access to precise control-flow graphs of the programs. This is trivial for <bigwig>, but certainly not for Java. Other future plans include type-safe support for XML document transformation, WML and VoiceXML support, and broadening the view towards development and management of whole Web sites comprising many services.

9.9.1 Acknowledgments

Tom Ball provided us with extensive and very helpful information about experiences with the MAWL language. Anders Sandholm was a key participant during his Ph.D. studies at BRICS. Mikkel Ricky Christensen and Steffan Olesen worked tirelessly as student programmers during the entire project. Niels Damgaard, Uffe Engberg, Mads Johan Jurik, Lone Haudrum Olesen, Christian Stenz, and Tommy Thorn provided valuable feedback and suggestions. We also appreciate the efforts made by the participants of the WIG Projects course in Spring 1998. Finally, we are grateful for the insightful comments we received from the anonymous reviewers.

Chapter 10

A Runtime System for Interactive Web Services

with Claus Brabrand, Anders Sandholm, and Michael I. Schwartzbach

Abstract

Interactive Web services are increasingly replacing traditional static Web pages. Producing Web services seems to require a tremendous amount of laborious low-level coding due to the primitive nature of CGI programming. We present ideas for an improved runtime system for interactive Web services built on top of CGI running on virtually every combination of browser and HTTP/CGI server. The runtime system has been implemented and used extensively in `<bigwig>`, a tool for producing interactive Web services.

10.1 Introduction

An interactive Web service consists of a global shared state (typically a database) and a number of distinct sessions that each contain some local private state and a sequential, imperative action. A Web client may invoke an individual thread of one of the given session kinds. The execution of this thread may interact with the client and inspect or modify the global state.

One way of providing a runtime system for interactive Web services would be to simply use plain CGI scripts [101]. However, being designed for much simpler tasks, the HTTP/CGI protocol by itself is inadequate for implementing the session concept. It neither supports long sessions involving many user interactions nor any kind of concurrency control. Being the only widespread standard for running Web services, this has become a serious stumbling stone in the development of complex modern Web services.

We present in this paper a runtime system built on top of the HTTP/CGI protocol that among other features has support for sessions and concurrency control. First, we motivate the need for a runtime system such as the one presented here. This is done by presenting its advantages over a simple CGI script based solution. Afterwards, a description of the runtime system, its different parts, and its dynamic behavior is given.

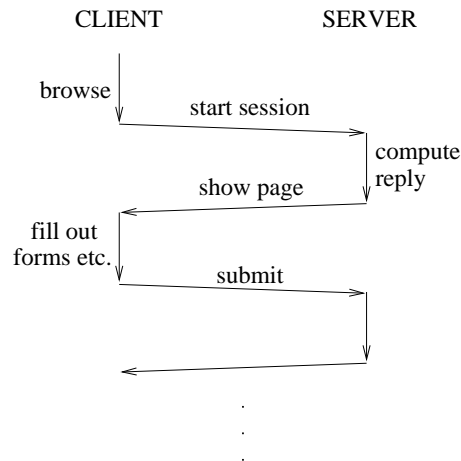


Figure 10.1: An interactive Web session

We round off with a discussion of related work, a conclusion, and directions for future work.

In the appendices, we briefly describe an implementation of the suggested runtime system. Also, we give a short presentation of <bigwig> [197], which is a tool for producing interactive Web services that makes extensive use of the self-contained runtime system package.

10.2 Motivation

The technology of plain CGI scripts lacks several of the properties one would expect from a modern programming environment. In the following we discuss various shortcomings of traditional CGI programming and motivate our solution to these problems, namely the design of an improved runtime system built on top of the standard HTTP/CGI protocol.

10.2.1 The Session Concept

First, we will describe and motivate the concept of an interactive Web service.

The HTTP protocol was originally designed for browsing *static* documents connected with hyperlinks. CGI together with forms allows *dynamic* creation of documents, that is, the contents of a document are constructed on the server at the time the document is requested. Dynamic documents have many advantages over static documents. For instance, the contents of the documents can be *tailor-made* and *always up-to-date*.

A natural extension of the dynamic-document model is the concept of *interactive* services, which is illustrated in Figure 10.1. Here the client does not browse a number of more or less independent statically or dynamically generated pages but is guided through a *session* controlled by a session thread on the server. This session can involve a number of user interactions. The session is initiated by the client submitting a “start session” request. The server then starts a thread controlling the new session. This

thread generates a reply page which is sent back to the client. The page typically contains some input fields that are filled in by the client. That information is sent to the server, which then generates the next reply, and so on, until the session terminates.

This session concept allows a large class of services to be defined. However, a number of practical problems needs to be solved in order to implement this model on top of the CGI model.

10.2.2 CGI Scripts and Sequential Session Threads

As explained above, a Web service session consists of a sequential computation that along the way presents information to the client and waits for replies. However, HTTP/CGI is a state-less protocol, meaning that execution of a CGI script only lasts until a page is shown to the Web client. This fact makes it rather tedious to program larger Web services involving many client interactions. The sequential computation has to be split up into the small bits of computation that happen between client interactions. Each of these small bits will then constitute a CGI script or an instance of a CGI call.

Furthermore, to achieve persistency of the local state, one has to store and restore it explicitly between CGI calls, for instance “hidden” in the Web page sent to the client. For simple services where the full session approach is not needed this stateless-server approach might be preferable, but it is clearly inadequate in general.

Thus, the problem of forced termination of the CGI script at each client interaction is two-fold:

- Having to deal with many small scripts makes the *writing* and *maintenance* of a Web service rather difficult because the control-flow of the service tends to become less clear from the program code.
- Starting up a whole new process every time a client interaction is performed is expensive in itself. On top of this a complete image of the local state has to be stored and restored each time a client interaction is required. The local state can potentially hold a lot of data, such as database contents. Thus one gets a *substantial overhead* in the execution of a Web service.

We provide a simple solution which splits CGI scripts into two components, namely *connectors* and *session threads*. A connector is a tiny transient CGI script that redirects input to a session thread, receives the response from that thread, and redirects it back to the Web client. The session threads are persistent processes running residently on the Web server. They survive CGI calls and can therefore implement a long sequential computation involving several client interactions. The use of transient connectors and persistent session threads decreases the difficulty of writing and maintaining Web services. Furthermore, it improves substantially on the overhead of the Web server during execution of a service.

10.2.3 Other CGI Shortcomings

Traditionally, reply pages from session threads are sent directly to the client. That is, the session thread (or the connector if using the system described above) writes the

page to standard-output and the Web server sends it on to the client browser. This basic approach imposes some annoying problems on the client:

- The client is not able to use “bookmarks” to identify the session, since selecting a bookmark might imply resending an old query to the server while the server expects a reply to a more recent interaction. It would be natural to the client if selecting a bookmarked session would continue the session from its current state. Obviously, this requires the server to always keep some kind of backup of the latest page sent to the client.
- In the session concept described in the previous section, it does not make sense to roll back execution of a session thread to a previous state. A thread can only be continued from its current point of execution. As a result of sending pages directly using the standard-output method, every new page shown to the client gets stacked up in the client’s browser. This means that the stack of visited pages becomes filled up with references to outdated pages. One result is that the “back” button in the browser becomes rather useless.

We suggest a simple solution where—instead of sending the reply itself—the session thread writes its reply to a file visible to the client and then sends to the client a *reference* to the reply file. By choosing the same URL for the duration of the session, this reference can then function as an identification of that particular session. This solves both the problem with bookmarks and with the “back” button. Pressing “back” will now bring the client back to the Web page where he started the session, which seems like a natural effect.

This method also opens up for an easy solution to another problem. Sometimes the server requires a long time to compute the information for the next page to be shown to the client. Naturally, the client may become *impatient* and lose interest in the service or assume that the server or the connection is down if no response is received within a certain amount of time. If confirmation in the form of a temporary response page is sent, the client will know that something is happening and that waiting will not be in vain.

This extra feature is implemented in the runtime system as follows. If a response is not ready within for instance 8 seconds, the connector responds with a reference to a temporary page (for instance saying “please wait”) and terminates. This page will then automatically be loaded by the clients Web browser and reload itself, say every 5 seconds. Once the session thread finishes its computation and the real response page is ready, the thread just replaces the temporary page with the real response page. This will have the effect that next time the page is reloaded, the real response page will be shown to the client.

This reloading can be done with standard HTML functionality. Of course the reloading causes some extra network traffic, but using this method is probably as close as one gets to server pushing in the world of CGI programming.

10.2.4 Handling Safety Requirements Consistently

Another serious problem with traditional CGI programming is that concurrency control, such as synchronization of sessions and locking of shared variables, gets handled

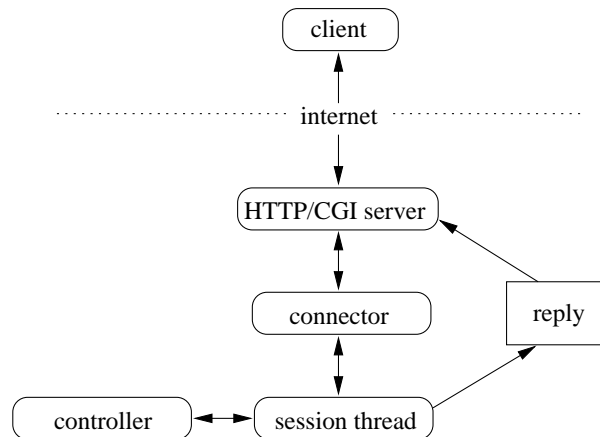


Figure 10.2: The runtime system

in an ad-hoc fashion. Typically, this is done using low-level semaphores supplied by the operating system.

As a result, Web services often implement these aspects incorrectly resulting in unstable execution and sometimes even damaging behavior.

Our solution allows one to put safety requirements, such as mutual exclusion or much more complex requirements, separately in a centralized supervising process called the controller. This approach significantly simplifies the job of handling safety requirements. Also, since each of the requirements can be formulated separately, the solution is much more robust towards changes in various parts of the code.

It is generally considered inefficient and unsafe to have centralized components in distributed systems. However, in this case the bottleneck is more likely to be the HTTP/CGI server and the network than the safety controller. In spite of that, we do try to distribute the functionality of our safety controller as discussed in Section 10.5.

10.3 Components in the Runtime System

At any time there will be a number of *Web clients* accessing the *HTTP/CGI server* through the HTTP/CGI protocol. On the server side we will have a *controller* and a number of *session threads* running. The session threads access the global data and produce response pages for the Web clients. From time to time a *connector* will be started as the result of a request from a Web client. The connector will make contact with the running session thread. A connector is shut down again after having delegated the answer from a session thread back to the Web client.

In the following we give a more detailed description of these components. For an overview of the components in the runtime system, see Figure 10.2.

Web clients Web clients are the users of the provided Web service. They make use of the service essentially by filling in forms and submitting HTTP/CGI requests using a browser.

The HTTP/CGI server The HTTP/CGI server handles the incoming HTTP/CGI requests by retrieving Web pages and starting up appropriate CGI scripts, in our case connectors. It also directs response pages back to the Web clients.

Session threads Session threads are the resident processes running on the Web server surviving several CGI calls. They represent the actual service code that implements the provided Web service. They do calculations, search databases, produce response Web pages, etc.

Connectors When a Web client makes a request through the server, a connector is started up. If this request is the first one made, the controller starts up a new session thread corresponding to the request made by the Web client. Otherwise—that is, if the Web client wants to continue execution of a running session thread—the connector notifies the relevant session thread that a request has been made and forwards the input to that thread.¹

Reply pages Each session thread has a designated file which contains the current Web page visible to the client of the session. When writing to this file, the whole contents is through a buffer updated atomically since the client may read the file at any time.

The controller The controller is a central component. It supervises session threads and has the possibility of suspending their execution at various points. This way it is ensured that the stated safety requirements are satisfied.²

Furthermore, the runtime system also contains a *global-state database* (could be the file-system or a full-fledged database), and a *service manager*, which takes care of garbage-collecting abandoned session threads and other administrative issues.

10.4 Dynamics of the Runtime System

In this section we describe the dynamic behavior of the runtime system. We start by explaining the overall structure of the execution of a session thread. Starting from this, we present each of the possible thread transitions.

First, it is described how a session thread is started. Then, transitions involving interaction with a Web client, that is, showing Web pages and getting replies, are dealt with. Finally, the transitions involving interaction with the controller are presented.

For each transition we give a description of the components involved and their interaction.

¹In the newest version of the runtime system implementation, we bypass the notions of CGI scripts and connector processes entirely. Instead, we use a specialized module for the Apache Web server, effectively moving the tasks of the connectors into the server itself. This improves performance substantially, since we do not create a new system process for every single interaction. For more information see [170].

²The controller process only appears in early versions of the implementation. As described in Section 3.6.1 we now rely on more standard techniques for performing concurrency control.

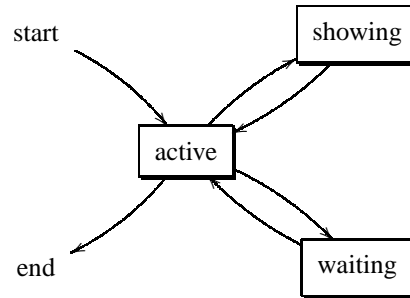


Figure 10.3: Possible states and transitions for a session thread

10.4.1 Execution of a Thread

The lifetime of a session thread is depicted in the diagram in Figure 10.3. When a thread is first started, it enters the state *active*. Here it can do all sorts of computations.

Eventually it reaches a point where it has composed a response HTML page. This page is shown to the Web client and the thread enters the state *showing*. Here it waits for the Web client to respond via yet another HTTP/CGI request. Upon re-submission the thread reenters the state *active* and resumes execution.

Note that in the world of naive CGI programming when moving from *active* to *showing* and back one would have to store a complete image of the local state before terminating the script. Then, when started again a new process would be started and the local state would have to be reconstructed from the image that was saved. This substantial overhead of saving and restoring local state is avoided completely by the use of transient connectors and resident threads.

While in state *active* a thread can get to a point in execution where safety critical computation, such as accessing a shared resource, needs to be carried out. When reaching such a point the thread asks the controller for permission to continue and enters the state *waiting*. When permission is granted from the controller the thread reenters the *active* state and continues execution.

With a traditional approach one would have to merge the code implementing the intricate details dealing with concurrency control with the service code. This inter-mixing would in addition to substantially reducing the readability of the code also increase the risk of introducing errors. Our solution separates the code dealing with concurrency control from the service code.

When the session is complete, the thread will leave the state *active* and end its execution.

10.4.2 Starting up a Session Thread

This section describes the transition from *start* to *active*.

When a new Web client makes an HTTP/CGI request, the server will start up a new connector as a CGI script. Since this request is the first one made by the Web client, a new thread is started according to the session name given in the request. As will be described later, a response page will be sent back to the client when the thread reaches a show call or a certain amount of time, for instance 8 seconds, has passed.

When a session thread is initiated or when it moves from *showing* to *active*, the contents of the reply file is immediately overwritten by a Web page containing a “reply not ready—please wait” message and a “refresh” HTML command. The “refresh” command makes the browser reload the page every few seconds until the temporary reply file is overwritten by the real reply as described in the following section. The default contents of the “please wait” page can be overridden by the service programmer by simply overwriting the reply file with a message more appropriate for the specific situation.

10.4.3 Interaction with the Client

During execution of a running thread the service can show a page to the Web client and continue execution when receiving response from the client. In the following we describe these two actions.

Showing a page

This section describes the transition from *active* to *showing*.

During execution of a session thread one can do computations, inspect the input from the client, produce response documents, etc. When a response document has been constructed and the execution reaches a point where the page is to be shown to the client, the following actions will be taken:

1. First, the document to be shown is written to the reply file as indicated in Figure 10.2. This file always contains a “no cache” pragma-command, so that the client browser always fetches a new page even though the same URL is used for the duration of the whole session. Unfortunately we thereby lose the possibility of browser caching, but being restricted to building on top of existing standards we cannot get it all.³
2. If the connector, that is, the CGI script started by the Web client, has not already terminated due to the 8 second timeout, the session thread tells it that the reply page is ready. After this, the thread goes to sleep.
3. When the connector either has been waiting the 8 seconds or it receives the “reply ready” signal from the session thread, the connector writes a location-reference containing the URL for the reply page onto standard-output (using the HTTP “location” feature), and then dies.
4. Finally, the HTTP/CGI server will transmit the URL back to the Web clients browser which then will fetch the reply page through the HTTP/CGI server and show it to the client.

In Figure 10.2, these actions describe a flow of data starting at the session thread and ending at the client.

³However, we have later discovered that the template-based mechanism for constructing Web pages used in <bigwig> makes it possible to cache the constant fragments that the pages are built from and thereby obtain the benefits of caching even for dynamically generated pages. This is described in [36].

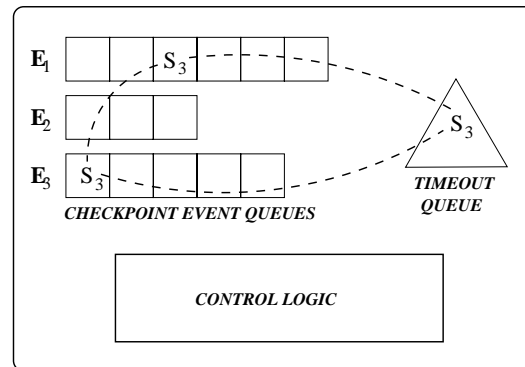


Figure 10.4: Components of the controller

Receiving client response

This section describes the transition from *showing* to *active*.

While the session thread is sleeping in the showing state, the Web client will read the page, fill out appropriate form fields, and resubmit. This will result in the following flow of data from the client to the session thread (see Figure 10.2):

1. First, a request is made by the client via the HTTP/CGI protocol. This request can be initiated either by clicking on a link or by pressing a submit button.
2. As a result, the HTTP/CGI server starts up a CGI script, that is, a connector.
3. The connector will then see that the client is already associated with a running thread and thus wake up that sleeping session thread and supply its new arguments.

10.4.4 Interaction with the Controller

The controller allows the programmer to restrict the execution of a Web service in such a way that stated safety requirements are satisfied.

Threads have built-in checkpoints at places where safety critical code is to be executed. At these checkpoints the thread must ask the controller for permission to continue. The controller, in turn, is constructed in such a way that it restricts execution according to the safety requirements and only allow threads that are not about to violate the requirements to continue.

In the following we describe in further detail the controller itself, what happens when session threads ask for permission, and how permission is granted by the controller.

The controller

The controller consists of three parts: some control logic, a number of checkpoint-event queues, and a timeout queue. Figure 10.4 gives an overview of the controller.

The control logic The control logic is the actual component representing the safety requirements. It controls whether events are enabled, and hence when the various session threads may continue execution at checkpoints. One could imagine various approaches, such as, the use of finite state machines or petri-nets. For that reason, the internals of the control logic are not specified here. The only requirement is that the interface must contain the following two functions available to the runtime system:

- `check_enabled` — takes a checkpoint-event ID as argument and replies whether that event is currently enabled.
- `event_occurred` — takes the ID of an enabled checkpoint-event as argument and updates the internal state of control logic with the information that the event has occurred.

We explain in the following how these functions are used in the controller.

Checkpoint-event queues The *checkpoint-event queues* form the interface to the running threads of the service. There is a queue for each possible checkpoint event. When a thread reaches a checkpoint it asks the controller for permission to continue by adding its process-ID onto the queues corresponding to the events it wants to wait for at the checkpoint.

Timeout queue As an extra feature one can specify a *timeout* when asking the controller for permission to continue. For this purpose the controller has a timeout queue. If permission is not granted within the specified time bound, the controller wakes up the thread with the information that permission has not been granted yet, but a timeout event has occurred. The specified timeouts are put in the special timeout queue (which is implemented as a priority queue).

Asking for permission at checkpoints

This section describes the transition from *active* to *waiting*.

As mentioned earlier, one has the possibility of adding checkpoints to session code where critical code is to be executed. The runtime system interface makes some functions available to the service programmer for specifying checkpoints. Conceptually, the programmer uses them to specify a “checkpoint statement” as illustrated with an example in Figure 10.5. This example would have the effect that whenever a thread instance of this session reaches this point it will do the following:

1. First, it will tell the controller that it waits for either an E_1 event, an E_3 event, or a timeout of 20 seconds.
2. Having sent this request to the controller, the thread goes to sleep waiting for a response.

Controller actions

When the controller is up and running, it loops doing the following:

```

wait {
  case E1:
    ...
  case E3:
    ...
  timeout 20:
    ...
}

```

Figure 10.5: A checkpoint example

- If it receives a request to pass a checkpoint from a client, the controller pushes the ID of the client onto the appropriate queues. These entries are chained so that later, when permission is granted, they can all be removed at once. Figure 10.4 illustrates the effect of the example from Figure 10.5 where entries belonging to a session, S_3 , are in the E_1 , E_3 and TIMEOUT queues.
- If a timeout has occurred, the controller deletes the affected entries in the queues and informs the involved thread.
- Otherwise, it will look for an enabled event using the `check_enabled` function from the control logic. If the queue corresponding to an enabled event is non-empty then the controller makes the event occur by doing the following:
 1. It removes the linked entries with the thread-ID of the enabled event from the respective queues,
 2. tells the control logic that the event has occurred using the `event_occurred` function, and
 3. wakes up the involved thread with a “permission granted” signal containing the name of the event.

If several events become enabled, a token-ring scheduling policy is used. This ensures fairness in the sense that if a thread waits for an enabled event, it will at some point be granted permission to continue.

Permission granted

This section describes the transition from *waiting* to *active*.

Having sent a request for permission to continue the thread is sleeping, waiting for the controller to make a response. If a “permission granted” signal is sent to the thread, it wakes up and continues, branching according to the event signaled by the controller. In the example checkpoint in Figure 10.5, if the controller grants permission for an E_1 event, execution is continued at the code following case E_1 . If the controller sends a “timeout” signal, execution continues after `timeout`.

10.5 Extending the Runtime System

The runtime system described in the previous sections can be extended in several ways. The following extensions either have been implemented in an experimental version of the runtime system package or will be in near future. With these extensions, we believe that we begin reaching the limits of what is possible with the standard HTTP/CGI protocol and the current functionality of standard browsers.

Distributed safety controller

To smoothen presentation, we have so far described the controller as one centralized component. In most cases it is possible to divide the control logic into independent parts controlling disjoint sets of checkpoint events. The controller can then be divided into a number of distributed control processes [194]. This way the problem of the controller being a bottleneck in the system is successfully avoided.

Service monitors

Using the idea of connectors and controllers, one can construct a “remote service monitor”, that is, a program run by a super-client, which is able to access logs and statistics information generated by the connectors and controllers, and to inspect and change the global state and the state of the control logic in the controllers. This can be implemented by having a dedicated *monitor process* for each service.

Secure communication

The system presented here is quite vulnerable to hostile attacks. It is easy to hijack a session, since the URL of the reply file is enough to identify a session. A simple solution is to use random keys in the URLs, making it practically impossible to guess a session ID. Of course, all information sent between the clients browser and the server, such as the session ID and all data written in forms, can still be eavesdropped. To avoid this, we have been doing experiments with cryptography, making all communication completely secure in practice. This requires use of browser plug-ins, which unfortunately has not been standardized. The protocols being used in the experiments are RSA, DES3, and RIPE-MD160. They prevent hijacking, provide secure channels, and verify user ID—all transparently to the client.⁴

Document clusters

In the session concept illustrated in Figure 10.1, only one page is generated and shown to the client at a time. However, often the service wants to generate a whole “cluster” of linked documents to the client and let the client browse these documents without involving the session thread. With the current implementation, a solution would be to program the possibility of browsing the cluster into the service code—inevitably a tedious and complicated task.

⁴Instead of using plug-ins we now apply the security techniques described in Section 3.6.2, in particular SSL and HTTP Authentication.

Document clusters can be implemented by simply having a reply file for each document in the cluster. Recall, however, that in the presented setup, the name of the reply file was fixed for the duration of a session. That way, the history buffer of the browser got a reasonable functionality. Therefore, to get that functionality we need a somewhat different approach: the reply files are not retrieved directly by the HTTP server but via a connector process. This connector receives the ID of the session thread in the CGI query string and the document number in a hidden variable.

Single process model

If all server processes (the session threads, safety controllers, etc.) are running on the same machine, that is, the possibility of distributing the processes is not being exploited, they might as well be combined into a single process using light-weight threads. This decreases the memory use (unless the operating system provides transparent sharing of code memory) and removes the overhead of process communication. The resulting system becomes something very close to being a dedicated Web server. The important difference being that it still builds upon the HTTP/CGI protocol.

10.6 Related Work

The idea of having persistent processes running residently on the server is central in the FastCGI [176] system. One difference is that FastCGI requires platform- and server-dependent support, while our approach works for all servers that support CGI. Also, our runtime system is tailored to support more specific needs.

A more detailed and formal description of how one can make use of safety requirements written separately in a suitable logic can be found in [194, 34]. A language for writing safety requirements is presented, the compilation process into a safety controller is described, and optimizations for memory usage and flow capacity of the controller are developed. A recent paper [113] generalizes these ideas resulting in a standard scheme for generating controllers for discrete event systems with both controllable and uncontrollable events.

The MAWL language [7, 68, 144] has been suggested as a domain-specific language for describing sequential transaction-oriented Web applications. Its high-level notation is also compiled into low-level CGI scripts. MAWL directly provides programming constructs corresponding to global state, dynamic document, sessions, local state, imperative actions, and client interactions. This system shows great promise to facilitate the efficient production of reliable Web services. While MAWL thus offers automatic synthesis of many advanced concepts, it still relies on standard low-level semaphore programming for concurrency control. Also, it does not have a FastCGI-like solution but instead it is possible to compile a service into a dedicated server for that particular service. Though being faster than using simple CGI scripts this solution is, as opposed to using a FastCGI-like solution, not easily ported between different machine architectures.

10.7 Conclusions and Future Work

The implementation as briefly described in the Appendix constitutes the core of the <bigwig> tool which currently is being developed at BRICS. In the <bigwig> tool, the runtime system we propose here has shown to provide simple and efficient solutions to problems occurring more and more often due to the increased use of interactive Web services. Furthermore, the session concept seems to constitute a framework which is very natural to use for designing complex services. By basing the design of the runtime system on very widely used protocols, the system is easy to incorporate. The further development of the runtime system can be followed on the <bigwig> homepage [197].

Appendix: Implementation

A UNIX version of the runtime system has been implemented (in C) as a package “runwig” containing the following components (corresponding to Figure 10.2):⁵

- The *connector*. It provides connection between the other components and the clients through the HTTP/CGI server.
- The *safety controller*, which handles synchronization and concurrency control. For the reasons described in Section 10.4.4, the control-logic is not included in the package but needs to be supplied separately.
- The *runtime library*, which is linked into the service code. It provides functions for easy interaction with the other components.

An experimental version of the runtime package implements the extensions described in Section 10.5. The runwig package—including all source code, detailed documentation, and examples—is available at <http://www.brics.dk/bigwig/runwig/>.

Appendix: <bigwig>

<bigwig> is a high-level programming language for developing interactive Web services. Complete specifications are compiled into a conglomerate of lower-level technologies such as CGI-scripts, HTML, JavaScript, Java applets, and plug-ins running on top the runtime system presented in this paper. <bigwig> is an intellectual descendant of the MAWL project but is a completely new design and implementation with vastly expanded ambitions.

The <bigwig> language is really a collection of tiny domain-specific languages focusing on different aspects of interactive Web services. To minimize the syntactic burdens, these contributing languages are held together by a C-like skeleton language. Thus, <bigwig> has the look and feel of C-programs with special data- and control-structures.

⁵As mentioned, in the latest version of runwig the notion of connectors has been integrated into the Web server and the safety controller has been replaced by other concurrency control mechanisms.

A `<bigwig>` service executes a dynamically varying number of threads. To provide a means of controlling the concurrent behavior, a thread may synchronize with a central controller that enforces the global behavior to conform to a regular language accepted by a finite-state automaton. That is, the “control logic” in `<bigwig>` consists of finite-state automata. The controlling automaton is not given directly, but is computed (by the MONA [131, 158] system) from a collection of individual concurrency constraints phrased in first-order logic. Extensions with counters and negated alphabet symbols add expressiveness beyond regular languages.

HTML documents are first-class values that may be computed and stored in variables. A document may contain named gaps that are placeholders for either HTML fragments or attributes in tags. Such gaps may at runtime be plugged with concrete values. Since those values may themselves contain further gaps, this is a highly dynamic mechanism for building documents. The documents are represented in a very compressed format, and the plug operations takes constant time only. A flow-sensitive type checker ensures that documents are used in a consistent manner.

A standard service executes with hardly any security. Higher levels of security may be requested, such that all communications are digitally signed or encrypted using 512 bit RSA and DES3. The required protocols are implemented using a combination of Java, JavaScript, and native plug-ins.⁶

The familiar struct and array data-structures are replaced with tuples and relations which allow for a simple construction of small relational databases. These are efficiently implemented and should be sufficient for databases no bigger than a few MBs (of which there are quite a lot). A relation may be declared to be external, which will automatically handle the connection to some external server. An external relation is accessed with (a subset of) the syntax for internal relations, which is then translated into SQL.

An important mechanism for gluing these components together is a fully general hygienic macro mechanism that allows `<bigwig>` programmers to extend the language by adding arbitrary new productions to its grammar. All nonterminals are potential arguments and result types for such macros that, unlike C-front macros, are soundly implemented with full alpha-conversions. Also, error messages remain sensible, since they are threaded back through macro expansion. This allows the definition of Very Domain-Specific Languages that contain specialized constructions for building chat rooms, shopping centers, and much more. Macros are also used to wrap concurrency constraints and other primitives in layers of user-friendly syntax.

Version 0.9 of `<bigwig>` is currently undergoing internal evaluation at BRICS.⁷ If you want to try it out, then contact us for more information. The documentation is very rough as yet, but this has a high priority in the next few months. The project is scheduled to deliver a version 1.0 of the `<bigwig>` tool in June 1999. This will be freely available in an open source distribution for UNIX.

⁶This has changed since the time of publication, as noted on p. 160.

⁷The latest release is version 2.0 and is available from the `runwig` home page [170]. A detailed list of changes can also be found there.

Chapter 11

PowerForms: Declarative Client-Side Form Field Validation

with Claus Brabrand, Mikkel Ricky, and Michael I. Schwartzbach

Abstract

All uses of HTML forms may benefit from validation of the specified input field values. Simple validation matches individual values against specified formats, while more advanced validation may involve interdependencies of form fields.

There is currently no standard for specifying or implementing such validation. Today, CGI programmers often use Perl libraries for simple server-side validation or program customized JavaScript solutions for client-side validation.

We present PowerForms, which is an add-on to HTML forms that allows a purely declarative specification of input formats and sophisticated interdependencies of form fields. While our work may be seen as inspiration for a future extension of HTML, it is also available for CGI programmers today through a preprocessor that translates a PowerForms document into a combination of standard HTML and JavaScript that works on all combinations of platforms and browsers.

The definitions of PowerForms formats are syntactically disjoint from the form itself, which allows a modular development where the form is perhaps automatically generated by other tools and the formats and interdependencies are added separately.

PowerForms has a clean semantics defined through a fixed-point process that resolves the interdependencies between all field values. Text fields are equipped with status icons that continuously reflect the validity of the text that has been entered so far, thus providing immediate feed-back for the user. For other GUI components the available options are dynamically filtered to present only the allowed values.

PowerForms are integrated into the <bigwig> system for generating interactive Web services, but is also freely available in an Open Source distribution as a stand-alone package.

11.1 Introduction

We briefly review some relevant aspects of HTML forms. The CGI protocol enables Web services to receive input from clients through forms embedded in HTML pages. An HTML form is comprised of a number of input fields each prompting the client for information.

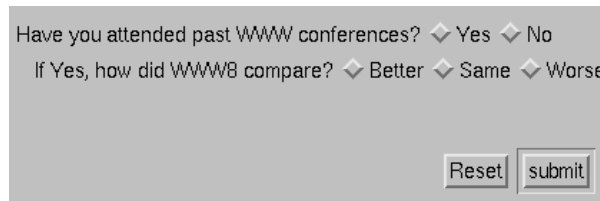
The visual rendering of an input field and how to enter the information it requests is determined by its type. The most widely used fields range from expecting lines of textual input to providing choices between a number of fixed options that were determined at the time the page was constructed. Many of the fields only differ in appearance and are indistinguishable to the server in the sense that they return the same kind of information. Fields of type `text` and `password`, although rendered differently, each expect one line of textual input from the client. Multiple lines of textual input can be handled through the `textarea` field. The fields of types `radio` and `select` both require exactly one choice between a number of static options, whereas an arbitrary number of choices are permitted by the `checkbox` and `select (multiple)` fields. Individual `radio` and `checkbox` fields with common name may be distributed about the form and constitute a group for which the selection requirements apply. The options of a `select` field, on the other hand, are grouped together in one place in the form. In addition, there are the more specialized fields, `image`, `file`, `button`, and `hidden`, which we shall not treat in detail. Finally, two fields control the behavior of the entire form, namely `reset` and `submit`, which respectively resets the form to its initial state and submits its contents to the server.

11.1.1 Input Validation

Textual input fields could possibly hold anything. Usually, the client is expected to enter data of a particular form, for instance a number, a name, a ZIP-code, or an e-mail address. The most frequent solution is to determine on the server whether the submitted data has the required form, which is known as *server-side input validation*. If some data are invalid, then those parts are presented once again along with suitable error messages, allowing the client to make the necessary corrections. This process is repeated until all fields contain appropriate data. This solution is simple, but it has three well-known drawbacks:

- it takes time;
- it causes excess network traffic; and
- it requires explicit server-side programming.

Note that these drawbacks affect all parties involved. The client is clearly annoyed by the extra time incurred by the round-trip to the server for validation, the server by the extra network traffic and “wasted” cycles, and the programmer by the explicit programming necessary for implementing the actual validation and re-showing of the pages. An obvious solution to the first two drawbacks is to move the validation from the server to the client, yielding *client-side input validation*. The third drawback, however, is only partially alleviated. All the details of re-showing pages are no longer required, but the actual validation still needs to be programmed.



Have you attended past WWW conferences? ☐ Yes ☐ No

If Yes, how did WWWB compare? ☐ Better ☐ Same ☐ Worse

Figure 11.1: Conference questionnaire.

The move from server-side to client-side also opens for another important benefit, namely the possibility of performing the validation *incrementally*. The client no longer needs to click the submit button before getting the validation report. This allows errors to be signalled as they occur, which clearly eases the task of correctly filling out the form.

11.1.2 Field Interdependencies

Another aspect of validation involves interdependent fields. Many forms contain fields whose values may be constrained by values entered in other fields. Figure 11.1 exhibits a simple questionnaire from a conference, in which participants were invited to state whether they have attended past conferences and if so, how this one compared. The second question clearly depends on the first, since it may only be answered if the first answer was positive. Conversely, an answer to the second question may be required if the first answer was “Yes”.

Such interdependencies are almost always handled on the server, even if the rest of the validation is addressed on the client-side. The reason is presumably that interdependencies require some tedious and delicate JavaScript code. This kind of validation is explicitly requested in the W3C working draft on extending forms [79]. One could easily imagine more advanced dependencies. Also, it would be useful if illegal selections could somehow automatically be deselected.

11.1.3 JavaScript Programming

Traditionally, client-side input validation is implemented in JavaScript. We will argue that this may not be the best choice for most Web authors.

First of all, using a general-purpose programming language for a relatively specific purpose exposes the programmer to many unnecessary details and choices. A small high-level domain-specific language dedicated to input validation would involve only relevant concepts and thus be potentially easier to learn and use. Many assisting libraries exist [172], but must still be used in the context of a full programming language.

Secondly, JavaScript code has an operational form, forcing the programmer to think about the order in which the fields and their contents are validated. However, the simplicity of the input validation task permits the use of a purely *declarative* approach. A declarative specification abstracts away operational details, making programs easier to read, write, and maintain. Also, such an approach is closer to composing HTML

than writing JavaScript, making input validation available to more people. As stated in the W3C working draft on design requirements for extending forms [79]:

“It will be possible to define a rich form, including validations, dependencies, and basic calculations without the use of a scripting language.”

Our solution will precisely include such mechanisms for validations and dependencies.

Finally, the traditional implementation task is further complicated by diverging JavaScript implementations in various browsers. This forces the programmer to stay within the subset of JavaScript that is supported by all browsers—a subset that may be hard to identify. In fact, a number of sites and FAQs are dedicated to identifying this subset [218, 143]. A domain-specific language could be compiled into this common subset of JavaScript, implying that only the compiler writer will be concerned with this issue.

11.1.4 Our Solution: PowerForms

As argued above, our solution is to introduce a high-level *declarative* and *domain-specific* language, called PowerForms, designed for incremental input validation.

Section 11.2 presents our solution for simple validation; Section 11.3 extends this to handle field interdependencies; Section 11.4 exhibits how other common uses of JavaScript also can be handled through declarative specification; Section 11.5 presents the overall strategy of the translation to JavaScript; and Section 11.6 describes the availability of the PowerForms packages.

11.1.5 Related Work

Authoring systems like ColdFusion [46] can automate server-side verification of some simple formats, but even so the result is unsatisfactory. A typical response to invalid data is shown in Figure 11.2. It refers to the internal names of input fields which are unknown to the client, and the required corrections must be remembered when the form is displayed again.

Active Forms [207] is based on a special browser supporting Form Applets programmed as Tcl scripts. It does not offer high-level abstractions or integration with HTML.

Web Dynamic Forms [97] offer an ambitious and complex solution. They propose a completely new form model that is technically unrelated to HTML and exists entirely within a Java applet. Inside this applet, they allow complicated interaction patterns controlled through an event-based programming model in which common actions are provided directly and others may be programmed in Java. When a form is submitted, the data are extracted from the applet and treated as ordinary HTML form data. The intervening years have shown that Web authors prefer to use standard HTML forms instead and then program advanced behavior in JavaScript. Thus, our simpler approach of automatically generating this JavaScript code remains relevant. An important reason to stay exclusively with HTML input fields is that they can be integrated into HTML tables to control their layout.

The XHTML-FML language [196] also provides a means for client-side input validation by adding an attribute called `cType` to textual input fields. However, this at-

Form Entries Incomplete or Invalid

One or more problems exist with the data you have entered.

- Data entered in the **Employees** field must be a number (you entered 'av').
- The value entered for the **Fulltextindexing** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Mmdirector** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Cgidatabase** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Hotjavaapplets** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Vrmlmodels** field must be between 1 and 10 (your entry was 33).
- The value entered for the **Adobeacrobat** field must be between 1 and 10 (your entry was 33).

Use the *Back* button on your web browser to return to the previous page and correct the listed problems.

Figure 11.2: Typical server-side validation.

tribute is restricted to a (large) set of predefined input validation types and there is no support for field interdependency.

Our PowerForms notation is totally declarative and requires no programming skills. Furthermore, it is modular in the sense that validation can be added to an input field in an existing HTML form without knowing anything but its name. The validation markup being completely separate from the form markup allows the layout of a form to be redesigned at any time in any HTML editor.

11.2 Validation of Input Formats

The language is based on regular expressions embedded in HTML that is subsequently translated into a combination of standard HTML and JavaScript. This approach benefits from an efficient implementation through the use of finite-state automata which are interpreted by JavaScript code.

Named formats may be associated to fields whose values are then required to belong to the corresponding regular sets. The client is continuously receiving feedback, and the form can only be submitted when all formats are satisfied. The server should of course perform a double-check, since the JavaScript code is open to tampering.

Regular expressions denoting sets of strings are a simple and familiar formalism for specifying the allowed values of form fields. As we will demonstrate, all reasonable input formats can be captured in this manner. Also, the underlying technology of finite-state automata gives a simple and efficient implementation strategy.

11.2.1 Syntax

We define a rich XML syntax [45] for regular expressions on strings:

$$\begin{aligned} regexp \rightarrow & \text{<const value=stringconst/>} \mid \\ & \text{<empty/>} \mid \\ & \text{<anychar/>} \mid \end{aligned}$$

```

<anything/> |
<charset value=stringconst/> |
<fix low=intconst high=intconst/> |
<relax low=intconst high=intconst/> |
<range low=charconst high=charconst/> |
<intersection> regexp* </intersection> |
<concat> regexp* </concat> |
<union> regexp* </union> |
<star> regexp </star> |
<plus> regexp </plus> |
<optional> regexp </optional> |
<repeat count=intconst> regexp </repeat>
<repeat low=intconst high=intconst> regexp </repeat>
<complement> regexp </complement> |
<regexp exp=stringconst/> |
<regexp id=stringconst> regexp </regexp> |
<regexp idref=stringconst/> |
<regexp uri=stringconst/> |
<include uri=stringconst/>

```

Here, *regexp** denotes zero or more repetitions of *regexp*. The nonterminals *stringconst*, *intconst*, and *charconst* have the expected meanings.

Note that the verbose XML syntax also allows standard Perl syntax for regular expressions through the construct `<regexp exp=stringconst/>`. Our full syntax is however more general, since it includes intersection, general complementation, import mechanisms, and a richer set of primitive expressions.

A regular expression is associated with a form field through a declaration:

```

formatdecl → <format name=stringconst
               help=stringconst
               error=stringconst>
               regexp
            </format>

```

The value of the optional *help* attribute will appear in the status line of the browser when the field has focus; similarly, the value of the optional *error* attribute will appear if the field contains invalid data.

The format takes effect for a form field of type `text`, `password`, `select`, `radio`, or `checkbox` whose name is the value of the *name* attribute. The need for input formats is perhaps only apparent for `text` and `password` fields, but we need the full generality later in Section 11.3.

11.2.2 Semantics of Regular Expressions

Each regular expression denotes an inductively defined set of strings. The *const* element denotes the singleton set containing its value. The empty element denotes the empty set. The *anychar* element denotes the set of all characters. The *anything* element denotes the set of all strings. The *charset* denotes the set of characters in its value. The *fix* element denotes the set of numerals from *low* to *high* all padded with leading zeros to have the same length as *high*. The *relax* element denotes the

set of numerals from `low` to `high`. The `range` element denotes the set of singleton strings obtained from the characters `low` to `high`. The `intersection` element denotes the intersection of the sets denoted by its children. The `concat` element denotes the concatenation of the sets denoted by its children. The `union` element denotes the union of the sets denoted by its children. The `star` element denotes zero or more concatenations of the set denoted by its child. The `plus` element denotes one or more concatenations of the set denoted by its child. The `optional` element denotes the union of the set containing the empty string and the set denoted by its child. The `repeat` element with attribute `count` denotes a fixed power of the set denoted by its child. The `repeat` element with attributes `low` and `high` denotes the corresponding interval of powers of the set denoted by its child, where `low` defaults to zero and `high` to infinity. The `complement` element denotes the complement of the set denoted by its child. The `regexp` element with attribute `exp` denotes the set denoted by its attribute value interpreted as a standard Perl regular expression. The `regexp` element with attribute `id` denotes the same set as its child, but in addition names it by the value of `id`. The `regexp` element with attribute `idref` denotes the same set as the regular expression whose name is the value of `idref`. It is required that each `id` value is unique throughout the document and that each `idref` value matches some `id` value. The `regexp` element with attribute `uri` denotes the set recognized by a precompiled automaton. The `include` element performs a textual insertion of the document denoted by its `url` attribute.

11.2.3 Semantics of Format Declarations

The effect on a form field of a regular expression denoting the set S is defined as follows. For a `text` or `password` field, the effect is to decorate the field with one of four annotations:

- *green light*, if the current value is a member of S ;
- *yellow light*, if the current value is a proper prefix of a member of S ;
- *red light*, if the current value is not a prefix of a member of a non-empty S ; or
- *n/a*, if S is the empty set.

The form cannot be submitted if it has a yellow or red light. The default annotations, which are placed immediately to the right of the field, are tiny icons inspired by traffic lights, but they can be customized with arbitrary images to obtain a different look and feel as indicated in Figure 11.3. Other annotations, like colorings of the input fields, would also seem reasonable, but current limitations in JavaScript make this impossible.

For a `select` field, the effect is to filter the `option` elements, allowing only those whose values are members of S . There is a slight deficiency in the design of a singular `select`, since it in some browser implementations will always show one selected element. To account for the situation where no option is allowed, we introduce an extension of standard HTML, namely `<option value="foo" error>` which is legal irrespective of the format. The form cannot be submitted if the `error` option is selected, unless S is the empty set.






	traffic	star	check	ok	blank
green light				OK	
yellow light		★			
red light		★			
n/a	NA			OK	

Figure 11.3: Different styles of status icons.

For a radio field, the effect is that the button can only be depressed if its value is a member of S ; if S is not the empty set, then the form cannot be submitted unless one button is depressed. Note that the analogue of the error option is the case where no button is depressed.

For a checkbox field, the effect is that the button can only be depressed if its value is a member of S .

Using our mechanism, it is possible to create a *deadlocked* form that cannot be submitted. The simplest example is the following, assuming the input field below is the only one in the radio button group named `foo`:

```
<input type="radio" name="foo" value="aaa">
<format name="foo"><const value="bbb"></format>
```

Regardless of whether the radio button `foo` is depressed or not, `foo` will never satisfy its requirements. Thus, the form can never be submitted. This behavior exposes a flaw in the design of the form, rather than an inherent problem with our mechanisms.

11.2.4 Examples

All reasonable data formats can be expressed as regular expressions, some more complicated than others. A simple example is the password format for user ID registration, seen in Figure 11.4, which is five or more characters not all alphabetic:

```
<regex id="pwd">
  <intersection>
    <repeat low="5"><anychar/></repeat>
    <complement>
      <star>
        <union>
          <range low="a" high="z"/>
          <range low="A" high="Z"/>
        </union>
      </star>
    </complement>
  </intersection>
```

Figure 11.4: User ID registration.

```

    </star>
  </complement>
</intersection>
</regexp>

```

or alternatively using the Perl syntax where possible:

```

<regexp id="pwd">
  <intersection>
    <regexp exp=".{5,}" />
    <complement>
      <regexp exp="[a-zA-Z]*" />
    </complement>
  </intersection>
</regexp>

```

To enforce this format on the existing form, we just add the declarations:

```

<format name="Password1"><regexp idref="pwd"/></format>
<format name="Password2"><regexp idref="pwd"/></format>

```

At our Web site we show more advanced examples, such as legal dates including leap days, URIs, and time of day. As a final example, consider a simple format for ISBN numbers:

```

<regexp id="isbn">
  <concat>
    <repeat count="9">
      <concat>
        <range low="0" high="9"/>
        <optional><charset value="-" /></optional>
      </concat>
    </repeat>
  </concat>
</regexp>

```

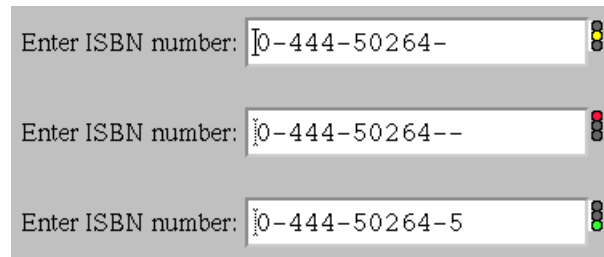


Figure 11.5: Checking ISBN numbers.

```

    <charset value="0123456789X"/>
  </concat>
</regexp>

```

or more succinctly:

```

<regexp id="isbn">
  <regexp exp="([0-9]([ -])?){9}[0-9X]"/>
</regexp>

```

An input field that exploits this format is:

```

Enter ISBN number: <input type="text" name="isbn" size=20>
<format name="isbn"
  help="Enter an ISBN number"
  error="Illegal ISBN format">
  <regexp idref="isbn"/>
</format>

```

Initially, the field has a yellow light. This status persists, as seen in Figure 11.5, while we enter the text "0-444-50264-" which is a legal prefix of an ISBN number. Entering another "-" yields a red light. Deleting this character and entering 5 will finally give a legal value and a green light.

While the input field has focus, the help string appears in the status line of the browser. If the client attempts to submit the form with invalid data in this field, then the error text appears in an alert box.

An ISBN format that includes checksums can be described as a complex regular expression that yields a 201-state automaton. This full format would only accept 5 as the last digit, since that is the correct checksum. Such a regular expression could hardly be written by hand; in fact, we generated it using a C program. But as precompiled automata may be saved and provided as formats, this shows that our technology also allows us to construct and publish a collection of advanced default formats, similarly to the datatypes employed in XML Schema [25] and the predefined ctype formats suggested in [196].

11.3 Interdependencies of Form Fields

We present a simple, yet general mechanism for expressing interdependencies. We have strived to develop a purely declarative notation that requires no programming

skills. Our proposal is based on dynamically evolving formats that are settled through a fixed-point process.

11.3.1 Syntax

We extend the syntax for formats as follows:

```

formatdecl → <format name=stringconst> format </format>

format      → regexp |
              <if> boolexp
                <then> format </then>
                <else> format </else>
              </if> |
              <format id=stringconst> format </format> |
              <format idref=stringconst/>

boolexp     → <match name=stringconst> regexp </match> |
              <equal name=stringconst value=stringconst/> |
              <and> boolexp* </and> |
              <or> boolexp* </or> |
              <not> boolexp* </not>

```

Now, the format that applies to a given field is dependent on the values of other fields. The specification is a binary decision tree, whose leaves are regular expressions and whose internal nodes are boolean expressions. Each boolean expression is a propositional combination of the primitive match and equal elements that each test the field indicated by name. Even this simple language is more advanced than required for most uses.

11.3.2 Semantics of Boolean Expressions

A boolean expression evaluates to true or false. For a text or password field, equal is true iff its current value equals value; match is true iff its current value is a member of the set denoted by regexp. For a select field, equal is true iff the value of a currently selected option equals value; match is true iff the value of a currently selected option is a member of the set denoted by regexp. For a collection of radio or checkbox fields, equal is true iff a button whose value equals value is currently depressed; match is true iff a button whose value is a member of the set denoted by regexp is currently depressed.

For the boolean operators, and is true iff all of its children are true, or is true if one of its children is true, and not is true if all of its children are false.

11.3.3 Semantics of Interdependencies

Given a collection of form fields F_1, \dots, F_n with associated formats and values, we define an *iteration* which in order does the following for each F_i :

- evaluate the current format based on the current values of all form fields;
- update the field based on the new current format.

The updating varies with the type of the form field:

- for a `text` field, the status light is changed to reflect the relationship between the current value and the current format;
- for a `select` field, the options are filtered by the new format, and the selected options that are no longer allowed by the format are unselected; if the current selection of a singular `select` is disallowed, the `error` option is selected;
- for a `radio` or `checkbox` field, a depressed button is released if its value is no longer allowed by the format.

An iteration is *monotonic*, which intuitively means that it can only delete user data. Technically, an iteration is a monotonic function on a specific lattice of form status descriptions. It follows that repeated iteration will eventually reach a fixed-point. In fact, if b is the total number of `radio` and `checkbox` buttons, p is the total number of `select` options, and s is the number of singular `selects`, then at most $b + p + s + 1$ iterations are required. Usually, however, the fixed-point will stabilize after very few iterations; also, a compile-time dependency analysis can keep this number down. Only complex forms with a high degree of interdependency will require many iterations.

The behavior of a PowerForm is to iterate to a new fixed-point whenever the client changes an input field; furthermore, the form data can only be submitted when all the form fields are in a status that allows this.

Note that the fixed-point we obtain is dependent on the order in which the form fields are updated: permuting the fields may result in a different fixed-point. We choose to update the fields in the textual order in which they appear in the document. This is typically the order in which the client is supposed to consider them, and the resulting fixed-point appears to coincide with the intuitively expected behavior. For simpler forms, the order is usually not significant.

With form interdependency it is not only possible to create a deadlocked form that can never be submitted, but also to create buttons that can never be depressed. Consider again the example from Section 11.2. Since the value `aaa` is different from `bbb`, the `foo` button will instantly be released whenever it is depressed. Such behavior can of course also stem from more complicated interdependent behavior.

The possible behaviors of PowerForms can in principle be analyzed statically. Define the size $|R|$ of a regular expression to be the number of states in the corresponding minimal, deterministic finite-state automaton, and the size $|F|$ of an input field to be the product of the sizes of all regular expressions that it may be tested against. Then a collection of input fields F_1, \dots, F_n determines a finite transition system with $|F_1||F_2| \cdots |F_n|$ states for which the reachability problem is decidable but hardly feasible in practice. We therefore leave it to the Web author to avoid aberrant behavior.

11.3.4 Examples

As a first example, we will redo the questionnaire from Figure 11.1:

```
Have you attended past WWW conferences?
☐

```

```

<br>
&nbsp;If Yes, how did WWW8 compare?
<input type="radio" name="compare" value="better">Better
<input type="radio" name="compare" value="same">Same
<input type="radio" name="compare" value="worse">Worse

```

To obtain the desired interdependence, we declare the following format:

```

<format name="compare">
  <if><equal name="past" value="yes"/>
    <then><complement><const value=""/></complement></then>
    <else><empty/></else>
  </if>
</format>

```

Only if the first question is answered in the positive, may the second group of radio buttons may be depressed and an answer is also required. A second example shows how radio buttons may filter the options in a selection:

```

Favorite letter group:
<input type="radio" name="group" value="vowel" checked>vowels
<input type="radio" name="group" value="consonant">consonants
<br>
Favorite letter:
<select name="letter">
  <option value="a">a
  <option value="b">b
  <option value="c">c
  ...
  <option value="x">x
  <option value="y">y
  <option value="z">z
</select>

```

The unadorned version of this form allows inconsistent choices such as group having value vowel and letter having value z. However, we can add the following format:

```

<format name="letter">
  <if><equal name="group" value="vowel"/>
    <then><charset value="aeiouy"/></then>
    <else><charset value="bcd fghjklmnpqrstvwxyz"/></else>
  </if>
</format>

```

Apart from enforcing consistency, the induced behavior will make sure that the client is only presented with consistent options, as shown in Figure 11.6. Next, consider the form:

```

<b>Personal info</b>
<p>
Name: <input type="text" name="name" size="30"><br>
Birthday: <input type="text" name="birthday" size="20"><br>

```

Favorite letter group: ☒ vowels ☐ consonants

Favorite letter: a
e
i
o
u
y

Figure 11.6: Only vowels are presented.

Personal info

Name:

Birthday:

Marital status: ☒ single
☐ married
☐ widow[er]

Spousal info

Name:

Deceased ☐

Figure 11.7: Collecting personal information.

```
<table border="0" cellpadding="0" cellspacing="0">
<tr><td valign="top">Marital status:</td>
<td><input type="radio" name="marital" value="single" checked>single
<br>
<input type="radio" name="marital" value="married">married
<br>
<input type="radio" name="marital" value="widow">widow[er]
</td>
</tr>
</table>
<p>
<b>Spousal info</b>
<p>
Name: <input type="text" name="spouse" size="30"><br>
Deceased <input type="radio" name="deceased" value="deceased">
```

Several formats can be used here. For the birthday, we select from our standard library a 35-state automaton recognizing legal dates including leap days:

```
<format name="birthday">
  <regexp uri="http://www.brics.dk/bigwig/powerforms/date.dfa"/>
</format>
```

Among the other fields, there are some obvious interdependencies. Spousal info is only relevant if the marital status is not single, and the spouse can only be deceased if the marital status is widow:

```
<format name="spouse">
  <if><equal name="marital" value="married"/>
    <then><regexp idref="handle"/></then>
  <else>
    <if><equal name="marital" value="single"/>
      <then><empty/></then>
      <else><regexp idref="handle"/></else>
    </if>
  </else>
</if>
</format>

<format name="deceased">
  <if><equal name="marital" value="widow"/>
    <then><const value="deceased"/></then>
    <else><empty/></else>
  </if>
</format>
```

Here, handle refers to some regular expression for the names of people. Note that if the marital status changes from widow to single, then the deceased button will automatically be released. Dually, it seems reasonable that after a change from single to widow, the deceased button should automatically be depressed. However, such action is generally not meaningful, since it may cause the form to oscillate between two settings. In our formalism, this would violate the monotonicity property that guarantees termination of the fixed-point iteration. Still, the form cannot be submitted until the deceased button is depressed for a widow. The initial form is shown in Figure 11.7.

An example of a more complex boolean expression involves the form in Figure 11.8. Here, simple formats determine that the correct style of phone numbers is used for the chosen country. The option of requesting a visit from the NYC office is only open to those customers who live in New York City. This constraint is enforced by the following format:

```
<format name="nyc">
  <if><and><equal name="country" value="US"/>
    <match name="phone">
      <concat>
        <union>
          <const value="212"/>
          <const value="347"/>
        </union>
      </concat>
    </match>
  </if>
```

Figure 11.8: Collecting customer information.

```

        <const value="646"/>
        <const value="718"/>
        <const value="917"/>
    </union>
    <anything/>
</concat>
</match>
</and>
<then><anything/></then>
<else><empty/></else>
</if>
</format>

```

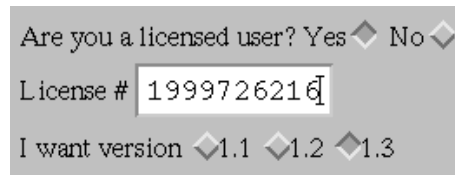
Residents from other cities will find that they cannot depress the button.

As a final example of the detailed control that we offer, consider the form in Figure 11.9 which invites users to request a new version of a product. Until the client has stated whether he has a license or not, it is impossible to choose a version. Once the choice has been made, licensed users can choose between all versions, others are limited to versions 1.1 and 1.2. The format on the last group of radio buttons is:

```

<format name="version">
  <if><equal name="license" value="yes"/>
    <then><anything/></then>
    <else>
      <if><equal name="license" value="no"/>
        <then><union>
          <const value="1.1"/>
          <const value="1.2"/>
        </union>
        </then>
        <else><empty/></else>
      </if>
    </else>
  </if>
</format>

```



Are you a licensed user? Yes ☐ No ☐

License #

I want version ☐ 1.1 ☐ 1.2 ☐ 1.3

Figure 11.9: Collecting user information.

11.4 Applet Results

Java applets can be used in conjunction with forms to implement new GUI components that collect data from the client. However, it is not obvious how to extract and validate data from an applet and submit it to the server on equal footing with ordinary form data.

We propose a simple mechanism for achieving this goal. We extend the applet syntax to allow `result` elements in addition to `param` elements. An example is the following:

```
<applet codebase="http://www.brics.dk/bigwig/powerapplets"
      code="sidebar.class">
  <param name="low" value="32">
  <param name="high" value="212">
  <result name="choice">
</applet>
```

When this applet is displayed, it shows a slide bar ranging over the interval [32..212]. When the form is submitted, the applet will be requested to supply a value for the `choice` result. This value is then assigned to a hidden form field named `choice` and will now appear with the rest of the form data. If the applet is not ready with the result, then the form cannot be submitted.

This extension only works for applets that are subclasses of the special class `PowerApplet` that we supply. It implements the method `putResult` that is used by the applet programmer to supply results, as well as the methods `resultsReady` and `getResult` that are called by the JavaScript code that implements the form submission.

In relation to `PowerForms`, applet results play the same role as input fields. Thus, they can have associated formats and be tested in boolean expressions. The value of an optional `error` attribute will appear in the alert box if an attempt is made to submit the form with a missing or invalid applet result.

11.5 Translation to JavaScript

A `PowerForms` document is parsed according to a very liberal HTML grammar that explicitly recognizes the special elements such as `format` and `regexp`. The generated HTML document retains most of the original structure, except that it contains the generated JavaScript code. Also, each input field is modified to include `onKeyUp`, `onChange`, and `onClick` functions that react to modifications from the client.

A function `update_foo` is defined for each input field name `foo`. This function checks if the current data is valid and reacts accordingly. Another function `update_all` is responsible for computing the global fixed-point.

Each regular expression is by the compiler transformed into a minimal, deterministic finite-state automaton, which is directly represented in a JavaScript data structure. It is a simple matter to use an automaton for checking if a data value is valid. For `text` and `password` fields, the status lights green, yellow, and red correspond to respectively an accept state, a non-accept state, and the crash state. For efficiency, the generated automata are time-stamped and cached locally; thus, they are only recompiled when necessary.

The generated code is quite small, but relies on a 500 line standard library with functions for manipulating automata and the Document Object Model [3].

11.6 Availability

The PowerForms system is freely available in an open source distribution from our Web site located at <http://www.brics.dk/bigwig/powerforms/>. The package includes documentation, the examples from this paper and many more, and the compiler itself which is written in 4000 lines of C. The generated JavaScript code has been tested for Netscape on Unix and Windows and for Explorer on Windows.

PowerForms are also directly supported by the `<bigwig>` system which is a high-level language for generating interactive Web services [40, 38, 195, 194]. It is likewise available at <http://www.brics.dk/bigwig/>.¹

11.7 Conclusion

We have shown how to enrich HTML forms with simple, declarative concepts that capture advanced input validation and field interdependencies. Such forms are subsequently compiled into JavaScript and standard HTML. This allows the design of more complex and interesting forms while avoiding tedious and error-prone JavaScript programming.

We would like to thank the entire `<bigwig>` team for assisting in experiments with PowerForms. Thanks also go to the PowerForms users, in particular Frederik Esser, for valuable feedback.

¹For a more up-to-date description of the features in PowerForms see [190] The newest version, which is based on Java and is incorporated into **JWIG**, is available from <http://www.brics.dk/~ricky/powerforms/>.

Chapter 12

Language-Based Caching of Dynamically Generated HTML

with Claus Brabrand, Steffan Olesen, and Michael I. Schwartzbach

Abstract

Increasingly, HTML documents are dynamically generated by interactive Web services. To ensure that the client is presented with the newest versions of such documents it is customary to disable client caching causing a seemingly inevitable performance penalty. In the <bigwig> system, dynamic HTML documents are composed of higher-order templates that are plugged together to construct complete documents. We show how to exploit this feature to provide an automatic fine-grained caching of document templates, based on the service source code. A <bigwig> service transmits not the full HTML document but instead a compact JavaScript recipe for a client-side construction of the document based on a static collection of fragments that can be cached by the browser in the usual manner. We compare our approach with related techniques and demonstrate on a number of realistic benchmarks that the size of the transmitted data and the latency may be reduced significantly.

12.1 Introduction

One central aspect of the development of the World Wide Web during the last decade is the increasing use of *dynamically* generated documents, that is, HTML documents generated using e.g. CGI, ASP, or PHP by a server at the time of the request from a client [221, 14]. Originally, hypertext documents on the Web were considered to be principally *static*, which has influenced the design of protocols and implementations. For instance, an important technique for saving bandwidth, time, and clock-cycles is to cache documents on the client-side. Using the original HTTP protocol, a document that never or rarely changes can be associated an “expiration time” telling the browsers and proxy servers that there should be no need to reload the document from the server before that time. However, for dynamically generated documents that change on every request, this feature must be disabled—the expiration time is always set to “now”, voiding the benefits of caching.

Even though most caching schemes consider all dynamically generated documents “non-cachable” [216, 15], a few proposals for attacking the problem have emerged [224, 186, 55, 116, 54, 78]. However, as described below, these proposals are typically not applicable for highly dynamic documents. They are often based on the assumptions that although a document is dynamically generated, 1) its construction on the server often does not have side-effects, for instance because the request is essentially a database lookup operation, 2) it is likely that many clients provide the same arguments for the request, or 3) the dynamics is limited to e.g. rotating banner ads. We take the next step by considering complex services where essentially every single document shown to a client is unique and its construction has side-effects on the server. A typical example of such a service is a Web-board where current discussion threads are displayed according to the preferences of each user. What we propose is not a whole new caching scheme requiring intrusive modifications to the Web architecture, but rather a technique for exploiting the caches already existing on the client-side in browsers, resembling the suggestions for future work in [221].

Though caching does not work for whole dynamically constructed HTML documents, most Web services construct HTML documents using some sort of constant templates that ideally ought to be cached, as also observed in [78, 220]. In Figure 12.1, we show a condensed view of five typical HTML pages generated by different `<bigwig>` Web services [40]. Each column depicts the dynamically generated raw HTML text output produced from interaction with each of our five benchmark Web services. Each non-space character has been colored either grey or black. The grey sections, which appear to constitute a significant part, are characters that originate from a large number of small, constant HTML templates in the source code; the black sections are dynamically computed strings of character data, specific to the particular interaction.

The lycos example simulates a search engine giving 10 results from the query “caching dynamic objects”; the bachelor service will based on a course roster generate a list of menus that students use to plan their studies; the jaoo service is part of a conference administration system and generates a graphical schedule of events; the webboard service generates a hierarchical list of active discussion threads; and the dmodlog service generates lists of participants in a course. Apart from the first simulation, all these examples are sampled from running services and use real data. The dmodlog example is dominated by string data dynamically retrieved from a database, as seen in Figure 12.1, and is thus included as a worst-case scenario for our technique. For the remaining four, the figure suggests a substantial potential gain from caching the grey parts.

The main idea of this paper is—automatically, based on the source code of Web services—to exploit this division into constant and dynamic parts in order to enable caching of the constant parts and provide an efficient transfer of the dynamic parts from the server to the client.

Using a technique based on JavaScript for shifting the actual HTML document construction from the server to the client, our contributions in this paper are:

- an automatic characterization, based on the source code, of document fragments as *cachable* or *dynamic*, permitting the standard browser caches to have significant effect even on dynamically generated documents;

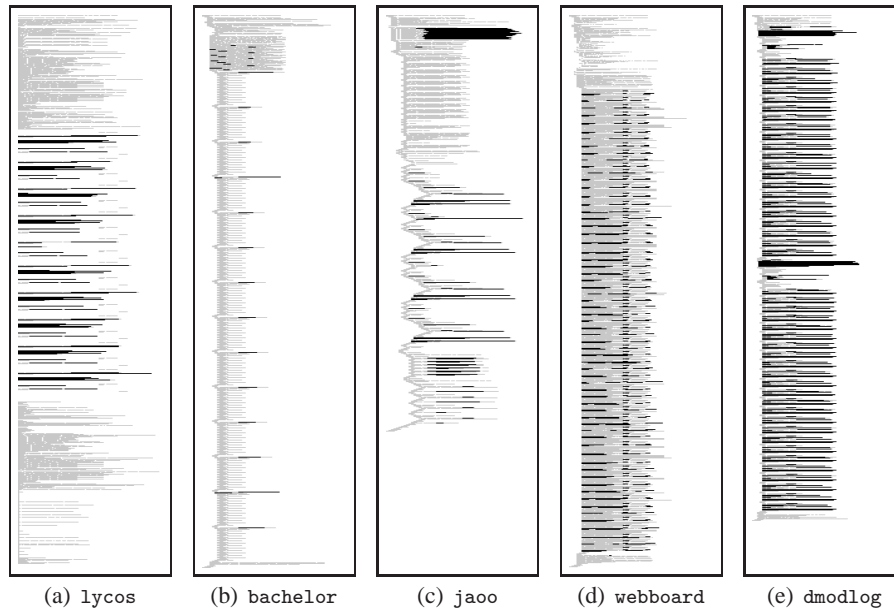


Figure 12.1: Benchmark services: cachable (grey) vs. dynamic (black) parts.

- a *compact representation* of the information sent to the client for constructing the HTML documents; and
- a generalization allowing a whole group of documents, called a *document cluster*, to be sent to the client in a single interaction and cached efficiently.

All this is possible and feasible due to the unique approach for dynamically constructing HTML documents used in the `<bigwig>` language [195, 40], which we use as a foundation. Our technique is non-intrusive in the sense that it builds only on preexisting technologies, such as HTTP and JavaScript—no special browser plug-ins, cache proxies, or server modules are employed, and no extra effort is required by the service programmer.

As a result, we obtain a simple and practically useful technique for saving network bandwidth and reviving the cache mechanism present in all modern Web browsers.

Outline

Section 12.2 covers relevant related work. In Section 12.3, we describe the `<bigwig>` approach to dynamic generation of Web documents in a high-level language using HTML templates. Section 12.4 describes how the actual document construction is shifted from server-side to client-side. In Section 12.5, we evaluate our technique by experimenting with five `<bigwig>` Web services. Finally, Section 12.6 contains plans and ideas for further improvements.

12.2 Related Work

Caching of dynamic contents has received increasing attention the last years since it became evident that traditional caching techniques were becoming insufficient. In the

following we present a brief survey of existing techniques that are related to the one we suggest.

Most existing techniques labeled “dynamic document caching” are either server-based, e.g. [186, 55, 116, 224], or proxy-based, e.g. [54, 199]. Ours is client-based, as e.g. the HPP language [78].

The primary goal for server-based caching techniques is not to lower the network load or end-to-end latency as we aim for, but to relieve the server by memoizing the generated documents in order to avoid redundant computations. Such techniques are orthogonal to the one we propose. The server-based techniques work well for services where many documents have been computed before, while our technique works well for services where every document is unique. Presumably, many services are a mixture of the two kinds, so these different approaches might support each other well—however, we do not examine that claim in this paper.

In [186], the service programmer specifies simple cache invalidation rules instructing a server caching module that the request of some dynamic document will make other cached responses stale. The approach in [224] is a variant of this with a more expressive invalidation rule language, allowing classes of documents to be specified based on arguments, cookies, client IP address, etc. The technique in [116] instead provides a complete API for adding and removing documents from the cache. That efficient but rather low-level approach is in [55] extended with *object dependency graphs*, representing data dependencies between dynamic documents and underlying data. This allows cached documents to be invalidated automatically whenever certain parts of some database are modified. These graphs also allow representation of *fragments* of documents to be represented, as our technique does, but caching is not on the client-side. A related approach for caching in the Weave Web site specification system is described in [222].

In [199], a protocol for proxy-based caching is described. It resembles many of the server-based techniques by exploiting equivalences between requests. A notion of *partial request equivalence* allows similar but non-identical documents to be identified, such that the client quickly can be given an approximate response while the real response is being generated.

Active Cache [54] is a powerful technique for pushing computation to proxies, away from the server and closer to the client. Each document can be associated a *cache applet*, a piece of code that can be executed by the proxy. This applet is able to determine whether the document is stale and if so, how to refresh it. A document can be refreshed either the traditional way by asking the server or, in the other extreme, completely by the proxy without involving the server, or by some combination. This allows tailor-made caching policies to be made, and—compared to the server-side approaches—it saves network bandwidth. The drawbacks of this approach are: 1) it requires installation of new proxy servers which can be a serious impediment to wide-spread practical use, and 2) since there is no general automatic mechanism for characterizing document fragments as cachable or dynamic, it requires tedious and error-prone programming of the cache applets whenever non-standard caching policies are desired.

Common to the techniques from the literature mentioned above is that truly dynamic documents, whose construction on the server often have side-effects and essentially always are unique (but contain common constant fragments), either cannot be

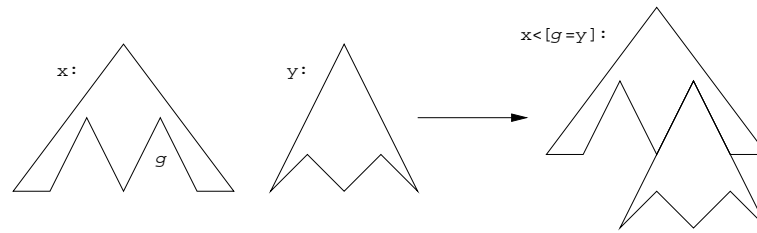
cached at all or require a costly extra effort by the programmer for explicitly programming the cache. Furthermore, the techniques either are inherently server-based, and hence do not decrease network load, or require installation of proxy servers.

Delta encoding [157] is based on the observation that most dynamically constructed documents have many fragments in common with earlier versions. Instead of transferring the complete document, a *delta* is computed representing the changes compared to some common base. Using a cache proxy, the full document is regenerated near the client. Compared to Active Cache, this approach is automatic. A drawback is—in addition to requiring specialized proxies—that it necessitates protocols for management of past versions. Such intrusions can obviously limit widespread use. Furthermore, it does not help with repetitions within a single document. Such repetitions occur naturally when dynamically generating lists and tables whose sizes are not statically known, which is common to many Web services that produce HTML from the contents of a database. Repetitions may involve both dynamic data from the database and static markup of the lists and tables.

The HPP language [78] is closely related to our approach. Both are based on the observation that dynamically constructed documents usually contain common constant fragments. HPP is an HTML extension which allows an explicit separation between static and dynamic parts of a dynamically generated document. The static parts of a document are collected in a *template* file while the dynamic parameters are in a separate *binding* file. The template file can contain simple instructions, akin to embedded scripting languages such as ASP, PHP, or JSP, specifying how to assemble the complete document. According to [78], this assembly and the caching of the templates can be done either using cache proxies or in the browser with Java applets or plug-ins, but it should be possible to use JavaScript instead, as we do.

Edge Side Includes [212] is an XML-based language for assembling HTML documents and other resources dynamically. The language is more expressive than the binding language in HPP, however all caching is performed by intermediate servers in a content delivery network, and not by the clients.

An essential difference between HPP and our approach is that the HPP solution is not integrated with the programming language used to make the Web service. With some work it should be possible to combine HPP with popular embedded scripting languages, but the effort of explicitly programming the document construction remains. Our approach is based on the source language, meaning that all caching specifications are automatically extracted from the Web service source code by the compiler and the programmer is not required to be aware of caching aspects. Regarding cachability, HPP has the advantage that the instructions describing the structure of the resulting document are located in the template file which is cached, while in our solution the equivalent information is in the dynamic file. However, in HPP the constant fragments constituting a document are collected in a single template. This means that HTML fragments that are common to different document templates cannot be reused by the cache. Our solution is more fine-grained since it caches the individual fragments separately. Also, HPP templates are highly specialized and hence more difficult to modify and reuse for the programmer. Being fully automatic, our approach guarantees cache soundness. Analogously to optimizing compilers, we claim that the <bigwig> compiler generates caching code that is competitive to what a human HPP programmer could achieve. This claim is substantiated by the experiments in Section 12.5. More-

Figure 12.2: The *plug* operator.

over, we claim that `<bigwig>` provides a more flexible, safe, and hence easier to use template mechanism than does HPP or any other embedded scripting language. The `<bigwig>` notion of *higher-order templates* is summarized in Section 12.3. A thorough comparison between various mechanisms supporting document templates can be found in [40].

As mentioned, we use compact JavaScript code to combine the cached and the dynamic fragments on the client-side. Alternatively, similar effects could be obtained using browser plug-ins or proxies, but implementation and installation would become more difficult. The HTTP 1.1 protocol [95] introduces both automatic compression using general-purpose algorithms, such as gzip, byte-range requests, and advanced cache-control directives. The compression features are essentially orthogonal to what we propose, as shown in Section 12.5. The byte-range and caching directives provide features reminiscent of our JavaScript code, but it would require special proxy servers or browser extensions to apply them to caching of dynamically constructed documents. Finally, we could have chosen Java instead of JavaScript, but JavaScript is more lightweight and is sufficient for our purposes.

12.3 Dynamic Documents in `<bigwig>`

The part of the `<bigwig>` Web service programming language that deals with dynamic construction of HTML documents is called DynDoc [195]. It is based on a notion of *templates* which are HTML fragments that may contain *gaps*. These gaps can at runtime be filled with other templates or text strings, yielding a highly flexible mechanism.

A `<bigwig>` *service* consists of a number of *sessions* which are essentially entry points with a sequential action that may be invoked by a client. When invoked, a session thread with its own local state is started for controlling the interactions with the client. Two built-in operations, *plug* and *show*, form the core of DynDoc. The *plug* operation is used for building documents. As illustrated in Figure 12.2, this operator takes two templates, x and y , and a gap name g and returns a copy of x where a copy of y has been inserted into every g gap. A template without gaps is considered a complete *document*. The *show* operation is used for interacting with the client, transmitting a given document to the client's browser. Execution of the client's session thread is suspended on the server until the client submits a reply. If the document contains input fields, the *show* statement must have a *receive* part for receiving the field values into program variables.

As in MAWL [144, 8], the use of templates permits programmer and designer

tasks to be completely separated. However, our templates are *first-class* values in that they can be passed around and stored in variables as any other data type. Also they are *higher-order* in that templates can be plugged into templates. In contrast, MAWL templates cannot be stored in variables and only strings can be inserted into gaps. The higher-order nature of our mechanism makes it more flexible and expressive without compromising runtime safety because of two compile-time program analyses: a *gap-and-field analysis* [195] and an *HTML validation analysis* [39]. The former analysis guarantees that at every *plug*, the designated gap is actually present at runtime in the given template and at every *show*, there is always a valid correspondence between the input fields in the document being shown and the values being received. The latter analysis will guarantee that every document being shown is valid according to the HTML specification. The following variant of a well-known example illustrates the DynDoc concepts:

```
service {
  html ask = <html>What? <input name="what"></html>;
  html hello = <html>Hello, <b><[thing]></b>!</html>;

  session HelloWorld() {
    string s;
    show ask receive [s=what];
    hello = hello<[thing=s]>;
    show hello;
  }
}
```

Two HTML variables, `ask` and `hello`, are initialized with constant HTML templates, and a session `HelloWorld` is declared. The entities `<html>` and `</html>` are merely lexical delimiters and are not part of the actual templates. When invoked, the session first shows the `ask` template as a complete document to the client. All documents are implicitly wrapped into an `<html>` element and a form with a default “continue” button before being shown. The client fills out the *what* input field and submits a reply. The session resumes execution by storing the field value in the `s` variable. It then plugs that value into the *thing* gap of the `hello` template and sends the resulting document to the client. The following more elaborate example will be used throughout the remainder of the paper:

```
service {
  html cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolor=[color]>
      <[contents]>
    </body>
  </html>;

  html greeting = <html>
    Hello <[who]>, welcome to <[what]>.
  </html>;

  html person = <html><i>Stranger</i></html>;
```



```

session welcome() {
  html h;
  h = cover<[color="#9966ff",
            contents=greeting<[who=person]]>;
  show h<[what=<html><b>BRICS</b></html>]>;
}
}

```

It builds a “welcome to BRICS” document by plugging together four constant templates and a single text string, shows it to the client, and terminates. The higher-order template mechanism does not require documents to be assembled bottom-up: gaps may occur non-locally as for instance the *what* gap in *h* in the *show* statement that comes from the *greeting* template being plugged into the *cover* template in the preceding statement. Its existence is statically guaranteed by the gap-and-field analysis.

We will now illustrate how our higher-order templates are more expressive and provide better cachability compared to first-order template mechanisms. First note that ASP, PHP, and JSP also fit the first-order category as they conceptually correspond to having one single first-order template whose special code fragments are evaluated on the server and implicitly plugged into the template. Consider now the unbounded hierarchical list of messages in a typical Web bulletin board. This is easily expressed recursively using a small collection of DynDoc templates. However, it can never be captured by any first-order solution without casting from templates to strings and hence losing type safety. Of course, if one is willing to fix the length of the list explicitly in the template at compile-time, it can be expressed, but not with unbounded lengths. In either case, sharing of repetitions in the HTML output is sacrificed, substantially cutting down the potential benefits of caching.

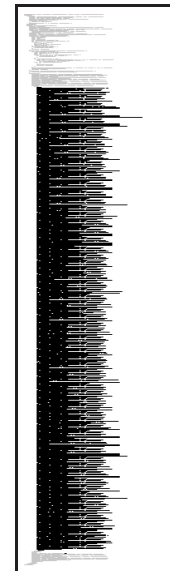


Figure 12.3 shows the webboard benchmark as it would appear if it had been generated entirely using first-order templates: only the outermost template remains and the message list is produced by one big dynamic area. Thus, nearly everything is dynamic (black) compared to the higher-order version displayed in Figure 12.1(d).

Languages without a template mechanism, such as Perl and C, that simply generate documents using low-level *print*-like commands generally have too little structure of the output to be exploited for caching purposes.

All in all, we have with the *plug-and-show* mechanism in *<bigwig>* successfully transferred many of the advantages known from static documents to a dynamic context. The next step, of course, being caching.

12.3.1 Dynamic Document Representation

Dynamic documents in *<bigwig>* are at runtime represented by the *DynDocDag* data structure supporting four operations: constructing constant templates, *constant(c)*;

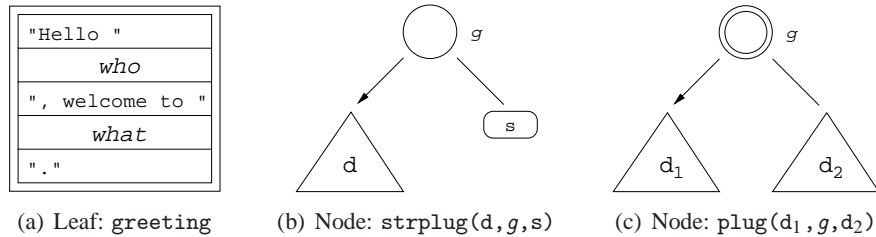


Figure 12.4: DynDocDag representation constituents.

string plugging, $\text{strplug}(d, g, s)$; template plugging, $\text{plug}(d_1, g, d_2)$; and showing documents, $\text{show}(d)$. This data structure represents a dynamic document as a binary DAG (Directed Acyclic Graph), where the leaves are either HTML templates or strings that have been plugged into the document and where the nodes represent pluggings that have constructed the document.

A constant template is represented as an ordered sequence of its text and gap constituents. For instance, the *greeting* template from the BRICS example service is represented as displayed in Figure 12.4(a) as a sequence containing two gap entries, *who* and *what*, and three text entries for the text around and between the gaps. A constant template is represented only *once* in memory and is shared among the documents it has been plugged into, causing the data structure to be a DAG in general and not a tree.

The string plug operation, strplug , combines a DAG and a constant string by adding a new string plug root node with the name of the gap, as illustrated in Figure 12.4(b). Analogously, the plug operation combines two DAGs as shown in Figure 12.4(c). For both operations, the left branch is the document containing the gap being plugged and the right branch is the value being plugged into the gap. Thus, the data structure merely records plug operations and defers the actual document construction to subsequent show operations.

Conceptually, the show operation is comprised of two phases: a *gap linking* phase that will insert a stack of links from gaps to templates and a *print traversal* phase that performs the actual printing by traversing all the gap links. The need for stacks comes from the template sharing.

The $\text{strplug}(d, g, s)$, $\text{plug}(d_1, g, d_2)$, and $\text{show}(d)$ operations have optimal complexities, $O(1)$, $O(1)$, and $O(|d|)$, respectively, where $|d|$ is the lexical size of the d document.

Figure 12.5 shows the representation of the document shown in the BRICS example service. In this simple example, the DAG is a tree since each constant template is used only once. Note that for some documents, the representation is exponentially more succinct than the expanded document. This is for instance the case with the following recursive function:

```
html tree(int n) {
  html list = <html><ul><li><[gap]><li><[gap]></ul></html>;
  if (n==0) return <html>foo</html>;
  return list<[gap]=tree(n-1)>;
}
```

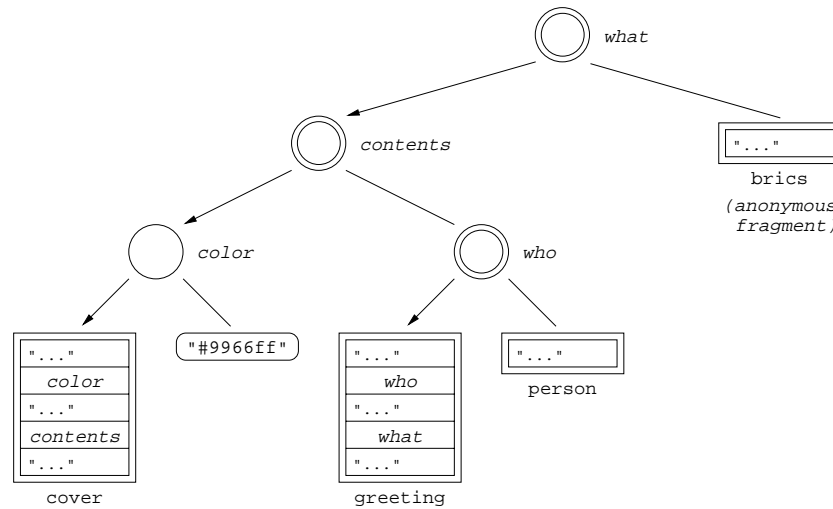


Figure 12.5: DynDocDag representation of the document shown in the BRICS example.

which, given n , in $O(n)$ time and space will produce a document of lexical size $O(2^n)$. This shows that regarding network load, it can be highly beneficial to transmit the DAG across the network instead of the resulting document, even if ignoring cache aspects.

12.4 Client-Side Caching

In this section we will show how to cache reoccurring parts of dynamically generated HTML documents and how to store the documents in a compact representation. The first step in this direction is to move the unfolding of the DynDocDag data structure from the server to the client. Instead of transmitting the unfolded HTML document, the server will now transmit a DynDocDag representation of the document in JavaScript along with a link to a file containing some generic JavaScript code that will interpret the representation and unfold the document on the client. Caching is then obtained by placing the constant templates in separate files that can be cached by the browser as any other files.

As we shall see in Section 12.5, both the caching and the compact representation substantially reduce the number of bytes transmitted from the server to the client. The compromise is of course the use of client clock cycles for the unfolding, but in a context of fast client machines and comparatively slow networks this is a sensible tradeoff. As explained earlier, the client-side unfolding is not a computationally expensive task, so the clients should not be too strained from this extra work, even with an interpreted language like JavaScript.

One drawback of our approach is that extra TCP connections are required for downloading the template files the first time, unless using the “keep connection alive” feature in HTTP 1.1. However, this is no worse than downloading a document with many images. Our experiments show that the number of transmissions per interaction is limited, so this does not appear to be a practical problem.

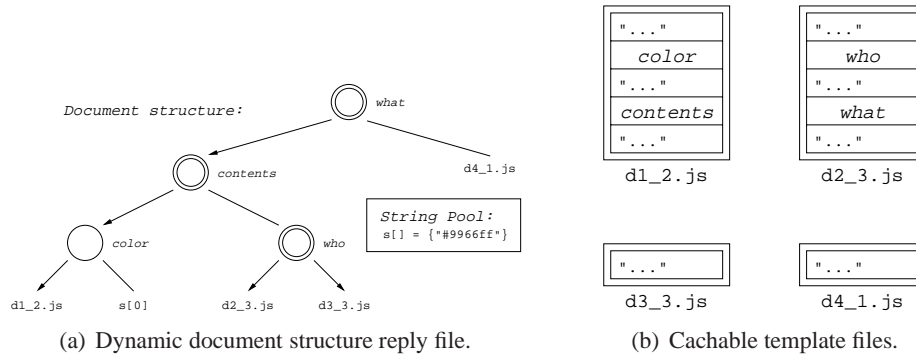


Figure 12.6: Separation into cachable and dynamic parts.

12.4.1 Caching

The DynDocDag representation has a useful property: it explicitly maintains a separation of the *constant templates* occurring in a document, the *strings* that are plugged into the document, and the *structure* describing how to assemble the document. In Figure 12.5, these constituents are depicted as framed rectangles, oval rectangles, and circles, respectively.

Experiments suggest that templates tend to occur again and again in documents shown to a client across the lifetime of a <bigwig> service, either because they occur 1) many times in the same document, 2) in many different documents, or 3) simply in documents that are shown many times. The strings and the structure parts, however, are typically dynamically generated and thus change with each document.

The templates account for a large portion of the expanded documents. This is substantiated by Figure 12.1, as earlier explained. Consequently, it would be useful to somehow cache the templates in the browser and to transmit only the dynamic parts, namely the strings and the structure at each *show* statement. This separation of cachable and dynamic parts is for the BRICS example illustrated in Figure 12.6.

As already mentioned, the solution is to place each template in its own file and include a link to it in the document sent to the client. This way, the caching mechanism in the browser will ensure that templates already seen are not retransmitted.

The first time a service shows a document to a client, the browser will obviously not have cached any of the JavaScript template files, but as more and more documents are shown, the client will download fewer and fewer of these files. With enough interactions, the client reaches a point of *asymptotic caching* where all constant templates have been cached and thus only the dynamic parts are downloaded.

Since the templates are statically known at compile-time, the compiler enumerates the templates and for each of them generates a file containing the corresponding JavaScript code. By postfixing template numbers with version numbers, caching can be enabled across recompilations where only some templates have been modified.

In contrast to HPP, our approach is entirely automatic. The distinction between static and dynamic parts and the DynDocDag structure are identified by the compiler, so the <bigwig> programmer gets the benefits of client-side caching without tedious and error-prone manual programming of bindings describing the dynamics.

12.4.2 Compact Representation

In the following we show how to encode the cachable template files and the reply documents containing the document representation. Since the reply documents are transmitted at each show statement, their sizes should be small. Decompression has to be conducted by JavaScript interpreted in browsers, so we do not apply general purpose compression techniques. Instead we exploit the inherent structure of the reply documents to obtain a lightweight solution: a simple yet compact JavaScript representation of the string and structure parts that can be encoded and decoded efficiently.

Constant Templates

A constant template is placed in its own file for caching and is encoded as a call to a JavaScript constructor function, `F`, that takes the number and version of the template followed by an array of text and gap constituents respectively constructed via calls to the JavaScript constructor functions `T` and `G`. For instance, the *greeting* template from the BRICS example gets encoded as follows:

```
F(T('Hello '),G(3),T(', welcome to '),G(4),T(''));
```

Assuming this is version 3 of template number 2, it is placed in a file called `d2_3.js`. The gap identifiers *who* and *what* have been replaced by the numbers 3 and 4, respectively, abstracting away the identifier names. Note that such a file needs only ever be downloaded once by a given client, and it can be reused every time this template occurs in a document.

Dynamics

The JavaScript reply files transmitted at each show contain three document specific parts: *include directives* for loading the cachable JavaScript template files, the *dynamic structure* showing how to assemble the document, and a *string pool* containing the strings used in the document.

The structure part of the representation is encoded as a JavaScript string constant, by a uuencode-like scheme which is tuned to the kinds of DAGs that occur in the observed benchmarks.

Empirical analyses have exposed three interesting characteristics of the strings used in a document: 1) they are all relatively short, 2) some occur many times, and 3) many seem to be URLs and have common prefixes. Since the strings are quite short, placing them in individual files to be cached would drown in transmission overhead. For reasons of security, we do not want to bundle up all the strings in cachable string pool files. This along with the multiple occurrences suggests that we collect the strings from a given document in a string pool which is inlined in the reply file sent to the client. String occurrences within the document are thus designated by their offsets into this pool. Finally, the common prefix sharing suggests that we collect all strings in a *trie* which precisely yields sharing of common prefixes. As an example, the following four strings:

```
"foo",
"http://www.brics.dk/bigwig/",
```

```
"http://www.brics.dk/bigwig/misc/gifs/bg.gif",
"http://www.brics.dk/bigwig/misc/gifs/bigwig.gif"
```

are linearized and represented as follows:

```
"foo|http://www.brics.dk/bigwig/[misc/gifs/b(igwig.gif|g.gif)]"
```

When applying the trie encoding to the string data of the benchmarks, we observe a reduction ranging from 1780 to 1212 bytes (on *bachelor*) to 27728 to 10421 bytes (on *dmodlog*).

The reply document transmitted to the client at the `show` statement in the BRICS example looks like:

```
<html>
<head>
  <script src="http://www.brics.dk/bigwig/dyndoc.js"></script>
  <script>I(1,2,3,4, 2,3,3,1);</script>
  <script>S("#9966ff"); D("/&E$I&I%",2,8,4);</script>
</head>
<body onload="E();"></body>
</html>
```

The document starts by including a generic 15K JavaScript library, `dyndoc.js`, for unfolding the DynDocDag representation. This file is shared among all services and is thus only ever downloaded once by each client as it is cached after the first service interaction. For this reason, we have not put effort into writing it compactly. The include directives are encoded as calls to the function `I` whose argument is an array designating the template files that are to be included in the document along with their version numbers. The `S` constructor function reconstructs the string trie which in our example contains the only string plugged into the document, namely “#9966ff”. As expected, the document structure part, which is reconstructed by the `D` constructor function, is not humanly readable as it uses the extended ASCII set to encode the dynamic structure. The last three arguments to `D` recount how many bytes are used in the encoding of a node, the number of templates plus plug nodes, and the number of gaps, respectively. The last line of the document calls the JavaScript function `E` that will interpret all constituents to expand the document. After this, the document has been fully replaced by the expansion. Note that three script sections are required to ensure that processing occurs in distinct phases and dependencies are resolved correctly. Viewing the HTML source in the browser will display the resulting HTML document, not our encodings.

Our compact representation makes no attempts at actual compression such as `gzip` or XML compression [149], but is highly efficient to encode on the server and to decode in JavaScript on the client. Compression is essentially orthogonal in the sense that our representation works independently of whether or not the transmission protocol compresses documents sent across the network, as shown in Section 12.5. However, the benefit factor of our scheme is of course reduced when compression is added.

12.4.3 Clustering

In `<bigwig>`, the `show` operation is not restricted to transmit a single document. It can be a collection of interconnected documents, called a *cluster*. For instance, a

document with input fields can be combined in a cluster with a separate document with help information about the fields.

A hypertext reference to another document in the same cluster may be created using the notation `&x` to refer to the document held in the HTML variable `x` at the time the cluster is shown. When showing a document containing such references, the client can browse through the individual documents without involving the service code. The control-flow in the service code becomes more clear since the interconnections can be set up as if the cluster were a single document and the references were internal links within it.

The following example shows how to set up a cluster of two documents, `input` and `help`, that are cyclically connected with `input` being the main document:

```
service {
  html input = <html>
    Please enter your name: <input name="name"><p>
    Click <a href=[help]>here</a> for help.
  </html>;

  html help = <html>
    You can enter your given name, family name, or nickname.
    <p><a href=[back]>Back</a> to the form.
  </html>;

  html output = <html>Hello <[name]>!</html>;

  session cluster_example() {
    html h, i;
    string s;
    h = help<[back=&i];
    i = input<[help=&h];
    show i receive [s=name];
    show output<[name=s];
  }
}
```

The cluster mechanism gives us a unique opportunity for further reducing network traffic. We can encode the entire cluster as a single JavaScript document, containing all the documents of the cluster along with their interconnections. Wherever there is a document reference in the original cluster, we generate JavaScript code to overwrite the current document in the browser with the referenced document of the cluster. Of course, we also need to add some code to save and restore entered form data when the client leaves and re-enters pages with forms. In this way, everything takes place in the client's browser and the server is not involved until the client leaves the cluster.

12.5 Experiments

Figure 12.7 recounts the experiments we have performed. We have applied our caching technique to the five Web service benchmarks mentioned in the introduction.

In Figure 12.7(b) we show the sizes of the data transmitted to the client. The grey columns show the original document sizes, ranging between 20 and 90 KB. The white

columns show the sizes of the total data that is transmitted using our technique, none of which exceeds 20 KB. Of ultimate interest is the black column which shows the asymptotic sizes of the transmitted data, when the templates have been cached by the client. In this case, we see reductions of factors between 4 and 37 compared to the original document size.

The lycos benchmark is similar to one presented for HPP [78], except that our reconstruction is of course in `<bigwig>`. It is seen that the size of our residual dynamic data (from 20,183 to 3,344 bytes) is virtually identical to that obtained by HPP (from 18,000 to 3,250 bytes). However, in that solution all caching aspects are hand-coded with the benefit of human insight, while ours is automatically generated by the `<bigwig>` compiler. The other four benchmarks would be more challenging for HPP.

In Figure 12.7(c) we repeat the comparisons from Figure 12.7(b) but under the assumption that the data is transmitted compressed using `gzip`. Of course, this drastically reduces the benefits of our caching technique. However, we still see asymptotic reduction factors between 1.3 and 2.9 suggesting that our approach remains worthwhile even in these circumstances. Clearly, there are documents for which the asymptotic reduction factors will be arbitrarily large, since large constant text fragments count for zero on our side of the scales while `gzip` can only compress them to a certain size. Hence we feel justified in claiming that compression is orthogonal to our approach. When the HTTP protocol supports compression, we represent the string pool in a naive fashion rather than as a trie, since `gzip` does a better job on plain string data. Note that in some cases our uncompressed residual dynamic data is smaller than the compressed version of the original document.

In Figure 12.7(d) and 12.7(e) we quantify the end-to-end latency for our technique. The total download and rendering times for the five services are shown for both the standard documents and our cached versions. The client is Internet Explorer 5 running on an 800 MHz Pentium III Windows PC connected to the server via either a 28.8K modem or a 128K ISDN modem. These are still realistic configurations, since by August 2000 the vast majority of Internet subscribers used dial-up connections [115] and this situation will not change significantly within the next couple of years [173]. The times are averaged over several downloads (plus renderings) with browser caching disabled. As expected, this yields dramatic reduction factors between 2.1 and 9.7 for the 28.8K modem. For the 128K ISDN modem, these factors reduce to 1.4 and 3.9. Even our “worst-case example”, `dmodlog`, benefits in this setup. For higher bandwidth dimensions, the results will of course be less impressive.

In Figure 12.7(f) we focus on the pure rendering times which are obtained by averaging several document accesses (plus renderings) following an initial download, caching it on the browser. For the first three benchmarks, our times are in fact a bit faster than for the original HTML documents. Thus, generating a large document is sometimes faster than reading it from the memory cache. For the last two benchmarks, they are somewhat slower. These figures are of course highly dependent on the quality of the JavaScript interpreter that is available in the browser. Compared to the download latencies, the rendering times are negligible. This is why we have not visualized them in Figure 12.7(d) and 12.7(e).

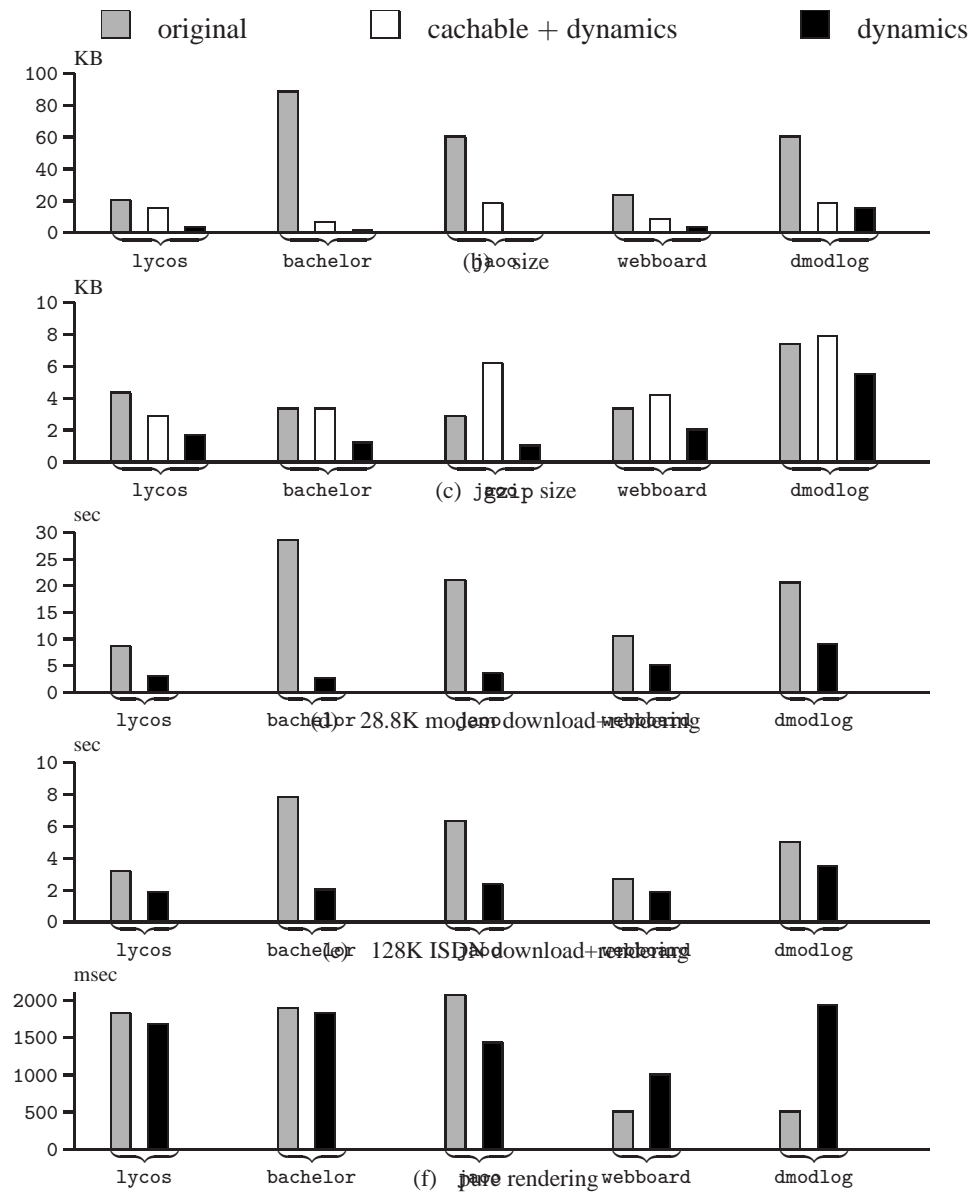


Figure 12.7: Experiments with the template representation.

12.6 Future Work

In the following, we describe a few ideas for further cutting down the number of bytes and files transmitted between the server and the client.

In many services, certain templates often occur together in all `show` statements. Such templates could be grouped in the same file for caching, thereby lowering the transmission overhead. In `<bigwig>`, the HTML validation analysis [39] already approximates a graph from which we can readily derive the set of templates that can reach a given `show` statement. These sets could then be analyzed for tightly connected templates using various heuristics. However, there are certain security concerns that need to be taken into consideration. It might not be good idea to indirectly disclose a template in a cache bundle if the `show` statement does not directly include it.

Finally, it is possible to also introduce language-based server-side caching which is complementary to the client-side caching presented here. The idea is to exploit the structure of `<bigwig>` programs to automatically cache and invalidate the documents being generated. This resembles the server-side caching techniques mentioned in Section 12.2.

12.7 Conclusion

We have presented a technique to revive the existing client-side caching mechanisms in the context of dynamically generated Web pages. With our approach, the programmer need not be aware of caching issues since the decomposition of pages into cachable and dynamic parts is performed automatically by the compiler. The resulting caching policy is guaranteed to be sound, and experiments show that it results in significantly smaller transmissions and reduced latency. Our technique requires no extensions to existing protocols, clients, servers, or proxies. We only exploit that the browser can interpret JavaScript code. These results lend further support to the unique design of dynamic documents in `<bigwig>`.

Chapter 13

Static Validation of Dynamically Generated HTML

with Claus Brabrand and Michael I. Schwartzbach

Abstract

We describe a static analysis of `<bigwig>` programs that efficiently decides if all dynamically computed XHTML documents presented to the client will validate according to the official DTD. We employ two data-flow analyses to construct a graph summarizing the possible documents. This graph is subsequently analyzed to determine validity of those documents. By evaluating the technique on a number of realistic benchmarks, we demonstrate that it is sufficiently fast and precise to be practically useful.

13.1 Introduction

Increasingly, HTML documents are dynamically generated by scripts running on a Web server, for instance using PHP, ASP, or Perl. This makes it much harder for authors to guarantee that such documents are really *valid*, meaning that they conform to the official DTD for HTML 4.01 or XHTML 1.0 [182]. Static HTML documents can easily be validated by tools made available by W3C and others. So far, the best possibility for a script author is to validate the dynamic HTML documents after they have been produced at runtime. However, this is an incomplete and costly process which does not provide any static guarantees about the behavior of the script. Alternatively, scripts may be restricted to use a collection of pre-validated templates, but this is generally not sufficiently expressive.

We present a novel technique for static validation of dynamic XHTML documents that are generated by a script. Our work takes place in the context of the `<bigwig>` language [40, 195], which is a full-fledged programming language for developing interactive Web services. In `<bigwig>`, XHTML documents are first-class citizens that are subjected to computations like all other data values. We instrument the compiler with an interprocedural data-flow analysis that extracts a grammatical structure, called a *summary graph*, covering the class of XHTML documents that a given program may

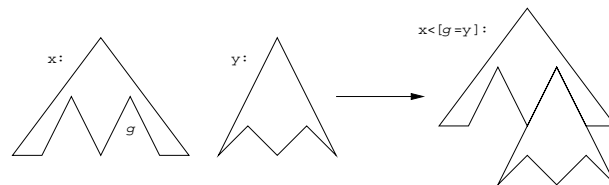
produce. Based on this information, the compiler statically determines if all documents in the given class conform to the DTD for XHTML 1.0. To accomplish this, we need to reformulate DTDs in a novel way that may be interesting in its own right. The analysis has efficiently handled all available examples. Furthermore, our technique can be generalized to more powerful grammatical descriptions.

13.1.1 Outline

First, in Section 13.2, we give a brief introduction to dynamically generating XHTML documents in the `<bigwig>` language. Section 13.3 formally defines the notion of summary graphs. In Sections 13.4 and 13.5, the two parts of the data-flow analysis are specified. Then, in Section 13.6, a notion of abstract DTDs is defined and used for specifying XHTML 1.0. Section 13.7 describes the algorithm for validating summary graphs with respect to abstract DTDs. In Section 13.8 we evaluate our implementation on ten `<bigwig>` programs. Finally, in Sections 13.9 and 13.10 we briefly describe related techniques and plans and ideas for future work.

13.2 XHTML Documents in `<bigwig>`

XHTML documents are just XML trees. In the `<bigwig>` language, XML *templates* are first-class data values that may be passed and stored as any other values. Templates are more general than XML trees since they may contain *gaps*, which are named placeholders that can be *plugged* with templates and strings: If x is an XML template with a gap named g and y is another XML template or a text string, then the plug operation, $x \ll [g=y]$, results in a new template which is copy of x where a copy of y has been inserted into the g gap:



A `<bigwig>` service consists of a number of *sessions*. A session thread can be invoked by a client who is subsequently guided through a number of interactions, controlled by the service code on the server. A *document* is a template where all gaps have been filled. When a complete XHTML document has been built on the server, it can be shown to the client who fills in the input fields, selects menu options, etc., and then continues the session by submitting the input to the session thread.

This plug-and-show mechanism provides a very expressive way of dynamically constructing Web documents. It is described in more detail in [195, 40] where a thorough comparison with other mechanisms is given and other aspects of `<bigwig>` are described. Since templates can be plugged into templates, these are *higher-order* templates, as opposed to the less flexible templates in the MAWL language [144, 8] where only strings can be plugged in.

Note that the number of gaps may both grow and shrink as the result of a plug operation. Also, gaps may appear in a non-local manner, as exemplified by the *what*

gap being plugged with the template BRICS in the following simple example in the actual <bigwig> syntax:

```
service {
  html cover = <html>
    <head><title>Welcome</title></head>
    <body bgcolor=[color]>
      <[contents]>
    </body>
  </html>;

  html greeting = <html>
    Hello <[who]>, welcome to <[what]>.
  </html>;

  html person = <html>
    <i>Stranger</i>
  </html>;

  session welcome() {
    html h;
    h = cover<[color="#9966ff",
              contents=greeting<[who=person]]>;
    show h<[what=<html><b>BRICS</b></html>]>;
  }
}
```

This service contains four constant templates and a session which when invoked will assemble a document using plug operations and show it to the client. Note that *color* is an *attribute gap* which can only be plugged with a string value, while the other gaps can also be plugged with templates. Constant templates are delimited by <html>...</html>. Implicitly, the mandatory surrounding <html> element is added to a document before being shown. Also, <head>, <title>, and <body> elements and a form with a default submit button is added if not already present. To simplify the presentation, we do not distinguish between HTML and XHTML since there are only minor syntactical differences. In the implementation, we allow HTML syntax but convert it to XHTML.

Note that <bigwig> is as general as all other languages for producing XML trees, since it is possible to define for each different element a tiny template like:

```
<html><ul style=[style]><[items]></ul></html>
```

that corresponds to a constructor function. The typical use of larger templates is mostly a convenience for the <bigwig> programmer.

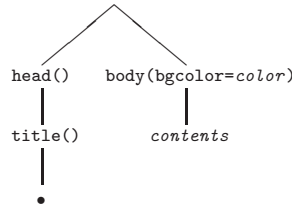
The <bigwig> compiler already contains an interprocedural data-flow analysis that keeps track of gaps and input fields in templates to enable type checking of plug and show operations [195]. That analysis statically ensures that the gaps are present when performing a plug operation and that the input fields in the documents being shown match the code that receives the values. However, the validity of the documents being shown has not been considered before, neither for <bigwig> or—to our knowledge—for any other programming language with such a flexible document construction mechanism.

13.2.1 XML Templates

We now formally define an abstract XML template. We are given an alphabet Σ of characters, an alphabet \mathbf{E} of element names, an alphabet \mathbf{A} of attribute names, an alphabet \mathbf{G} of template gap names, and an alphabet \mathbf{H} of attribute gap names. For simplicity, all alphabets are assumed to be disjoint. An *XML template* is generated by Φ in the following grammar:

$$\begin{aligned}
 \Phi &\rightarrow \varepsilon \\
 &\rightarrow \bullet \\
 &\rightarrow g && g \in \mathbf{G} \\
 &\rightarrow e(\Delta)\Phi && e \in \mathbf{E} \\
 &\rightarrow \Phi_1\Phi_2 \\
 \Delta &\rightarrow \varepsilon \\
 &\rightarrow (a = s) && a \in \mathbf{A}, s \in \Sigma^* \\
 &\rightarrow (a = h) && a \in \mathbf{A}, h \in \mathbf{H} \\
 &\rightarrow \Delta_1\Delta_2
 \end{aligned}$$

An XML template is a list of ordered trees where the internal nodes are *elements* with *attributes* and the leaves are either empty nodes, *character data* nodes, or *gap* nodes. Element attributes are generated by Δ . The \bullet symbol represents an arbitrary sequence of character data. We ignore the actual data, since those are never constrained by DTDs, unlike attribute values which we accordingly represent explicitly. As an example, we view the cover template abstractly as follows if we ignore character data nodes consisting only of white-space:



We introduce a function:

$$gaps : (\Phi \cup \Delta) \rightarrow 2^{\mathbf{G} \cup \mathbf{H}}$$

which gives the set of gap names occurring in a template or attribute list:

$$\begin{aligned}
 gaps(\varepsilon) &= \emptyset \\
 gaps(\bullet) &= \emptyset \\
 gaps(g) &= \{g\} \\
 gaps(e(\delta)\varphi) &= gaps(\delta) \cup gaps(\varphi) \\
 gaps(\varphi_1\varphi_2) &= gaps(\varphi_1) \cup gaps(\varphi_2) \\
 gaps(a = s) &= \emptyset \\
 gaps(a = h) &= \{h\} \\
 gaps(\delta_1\delta_2) &= gaps(\delta_1) \cup gaps(\delta_2)
 \end{aligned}$$

A template φ with a unique root element and with $gaps(\varphi) = \emptyset$ is considered a complete *document*.

13.2.2 Programs

We represent a <bigwig> program abstractly as a control-flow graph with atomic statements at each program point. The actual syntax for <bigwig> is very liberal and resembles C or Java code with control structures and functions. For <bigwig> it is a simple task to extract the normalized representation. If the underlying language had a richer control structure, for instance with inheritance and virtual methods or higher-order functions, we would need a preliminary control-flow analysis to provide the control-flow graph.

A program uses a set X of XML template variables and a set Y of string variables. The atomic statements are:

$x_i = x_j;$	(template variable assignment)
$x_i = \phi;$	(template constant assignment)
$y_i = y_j;$	(string variable assignment)
$y_i = s;$	(string constant assignment)
$y_i = \bullet;$	(arbitrary string assignment)
$x_i = x_j < [g=x_k];$	(template gap plugging)
$x_i = x_j < [h=y_k];$	(attribute gap plugging)
show $x_i;$	(client interaction)

where $x \in X$ and $y \in Y$ for each x and y . The assignments have the obvious semantics. The plug statement replaces all occurrences of a named gap with the given value. The show statement implicitly plugs all remaining gaps with ϵ before the template is displayed to the client. Also, the template is implicitly plugged into a wrapper template like the following:

```
<html>
  <head><title></title></head>
  <body>
    <form action="...">
      <[doc]>
        <input type="submit" value="continue">
      </form>
    </body>
  </html>
```

for completing the document and adding a “continue” button. The <head>, <title>, <body>, and <input> elements are of course only added if not already present. Since we here ignore input fields in documents, the receive part of show statements is omitted in this description.

13.3 Summary Graphs

Given a program control-flow graph, we wish to extract a finite representation of all the templates that can possibly be constructed at runtime. A program contains a finite collection of constant XML templates that are identified through a mapping function:

$$f : \mathbf{N} \rightarrow \Phi$$

where \mathbf{N} is the finite set of indices of the templates occurring in the program. A program also contains a finite collection of string constants, which we shall denote by $\mathcal{C} \subseteq \Sigma^*$. We now define a *summary graph* as a triple:

$$G = (R, E, \alpha)$$

where $R \subseteq \mathbf{N}$ is a set of *roots*, $E \subseteq \mathbf{N} \times \mathbf{G} \times \mathbf{N}$ is a set of *edges*, and $\alpha : \mathbf{N} \times \mathbf{H} \rightarrow \mathcal{S}$ is an attribute labeling function, where $\mathcal{S} = 2^{\mathcal{C}} \cup \{\bullet\}$. Intuitively, \bullet denotes the set of all strings.

Each summary graph G defines a set of XML templates, which is called the *language* of G and is denoted $L(G)$. Intuitively, this set is obtained by unfolding the graph from each root while performing all possible pluggings enabled by the edges and the labeling function. Formally, we define:

$$L(G) = \{\varphi \in \Phi \mid \exists r \in R : G, r \vdash \mathbf{f}(r) \Rightarrow \varphi\}$$

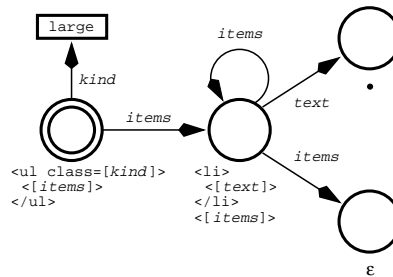
where the derivation relation \Rightarrow is defined for templates as:

$$\begin{array}{c} \frac{}{G, n \vdash \varepsilon \Rightarrow \varepsilon} \quad \frac{}{G, n \vdash \bullet \Rightarrow \bullet} \\[10pt] \frac{(n, g, m) \in E \quad G, m \vdash \mathbf{f}(m) \Rightarrow \varphi}{G, n \vdash g \Rightarrow \varphi} \\[10pt] \frac{G, n \vdash \delta \Rightarrow \delta' \quad G, n \vdash \varphi \Rightarrow \varphi'}{G, n \vdash e(\delta)\varphi \Rightarrow e(\delta')\varphi'} \\[10pt] \frac{G, n \vdash \varphi_1 \Rightarrow \varphi'_1 \quad G, n \vdash \varphi_2 \Rightarrow \varphi'_2}{G, n \vdash \varphi_1\varphi_2 \Rightarrow \varphi'_1\varphi'_2} \end{array}$$

and for attribute lists as:

$$\begin{array}{c} \frac{\alpha(n, h) \neq \bullet \quad s \in \alpha(n, h)}{G, n \vdash (a = h) \Rightarrow (a = s)} \\[10pt] \frac{\alpha(n, h) = \bullet \quad s \in \Sigma^*}{G, n \vdash (a = h) \Rightarrow (a = s)} \\[10pt] \frac{G, n \vdash \delta_1 \Rightarrow \delta'_1 \quad G, n \vdash \delta_2 \Rightarrow \delta'_2}{G, n \vdash \delta_1\delta_2 \Rightarrow \delta'_1\delta'_2} \end{array}$$

As an example, consider the following summary graph consisting of four template nodes, four plug edges, and a single attribute labeling:



Template nodes, root nodes, and attribute labels are drawn as circles, double circles, and boxes, respectively. The language of this summary graph is the set of all ul lists of class large with one or more character data items.

13.4 Gap Track Analysis

To obtain sufficient precision of the actual validation analysis, we first perform an initial analysis that tracks the origins of gaps. We show in Section 13.5 exactly why this information is necessary.

13.4.1 Lattices

The lattice for this analysis is simply:

$$T = (\mathbf{G} \cup \mathbf{H}) \rightarrow 2^{\mathbf{N}}$$

ordered by pointwise subset inclusion. For each program point ℓ we wish to compute an element of the derived lattice:

$$\text{TrackEnv}_\ell : X \rightarrow T$$

which inherits its structure from T . Intuitively, an element of this lattice tells us for a given variable x and a gap name g whether or not g can occur in the value of x , and if it can, which constant templates g can originate from.

13.4.2 Transfer Functions

Each atomic statement defines a transfer function $\text{TrackEnv}_\ell \rightarrow \text{TrackEnv}_\ell$ which models its semantics in a forward manner. If the argument is χ , then the results of applying this transfer function are:

$$\begin{array}{ll} x_i = x_j ; & \chi[x_i \mapsto \chi(x_j)] \\ x_i = \phi ; & \chi[x_i \mapsto \text{tfrag}(\phi, n)], \text{ where } \phi \text{ has index } n \\ x_i = x_j < [g=x_k] ; & \chi[x_i = \text{tplug}(\chi(x_j), g, \chi(x_k))] \\ x_i = x_j < [h=y_k] ; & \chi[x_i = \text{tplug}(\chi(x_j), h, \lambda p. \emptyset)] \end{array}$$

where we make use of some auxiliary functions:

$$\text{tfrag}(\phi, n) = \lambda p. \text{if } p \in \text{gaps}(\phi) \text{ then } \{n\} \text{ else } \emptyset$$

$$\text{tplug}(\tau_1, p, \tau_2) = \lambda q. \text{if } p=q \text{ then } \tau_2(q) \text{ else } \tau_1(q) \cup \tau_2(q)$$

For the remaining statement types, the transfer function is the identity function. The *tfrag* function states that all gaps in the given template originates from just that template. The *tplug* function adds all origins from the template being inserted and removes the existing origins for the gap being plugged.

13.4.3 The Analysis

It is easy to see that all transfer functions are monotonic, so we can compute the least fixed point iteratively in the usual manner [174]. The end result is for each program point ℓ an environment $track_\ell : X \rightarrow T$, which we use in the following as a conservative, upper approximation of the origins of the gaps. We omit the proof of correctness.

13.5 Summary Graph Analysis

We wish to compute for every program point and for every variable a summary of its possible values. A set of XML templates is represented by a summary graph and a set of string values by an element of S .

13.5.1 Lattices

To perform a standard data-flow analysis, we need both of these representations to be lattices. The set S is clearly a lattice, ordered by set inclusion and with \bullet as a top element. The set of summary graphs, called G , is also a lattice with the ordering defined by:

$$G_1 \sqsubseteq G_2 \Leftrightarrow R_1 \subseteq R_2 \wedge E_1 \subseteq E_2 \wedge \alpha_1 \sqsubseteq \alpha_2$$

where the ordering on S is lifted pointwise to labeling functions α . Clearly, both S and G are finite lattices. For each program point we wish to compute an element of the derived lattice:

$$Env_\ell = (X \rightarrow G) \times (Y \rightarrow S)$$

which inherits its structure from the constituent lattices.

13.5.2 Transfer Functions

Each atomic statement defines a transfer function $Env_\ell \rightarrow Env_\ell$, which models its semantics. If the argument is the pair of functions (χ, γ) and ℓ is the entry program point of the statement, then the results are:

$$\begin{array}{ll} x_i = x_j ; & (\chi[x_i \mapsto \chi(x_j)], \gamma) \\ x_i = \phi ; & (\chi[x_i \mapsto frag(n)], \gamma), \text{ where } \phi \text{ has index } n \\ y_i = y_j ; & (\chi, \gamma[y_i \mapsto \gamma(y_j)]) \\ y_i = s ; & (\chi, \gamma[y_i \mapsto \{s\}]) \\ y_i = \bullet ; & (\chi, \gamma[y_i \mapsto \bullet]) \\ x_i = x_j < [g=x_k] ; & (\chi[x_i \mapsto gplug(\chi(x_j), g, \chi(x_k), \\ & \quad track_\ell(x_j))], \gamma) \\ x_i = x_j < [h=y_k] ; & (\chi[x_i \mapsto hplug(\chi(x_j), h, \gamma(y_k), \\ & \quad track_\ell(x_j))], \gamma) \\ \text{show } x_i ; & (\chi, \gamma) \end{array}$$

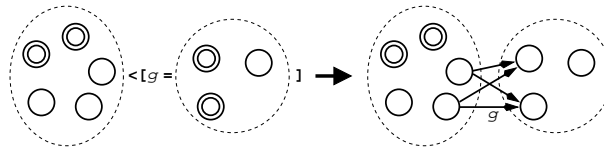
where we make use of some auxiliary functions:

$$\text{frag}(n) = (\{n\}, \emptyset, \lambda(m, h). \emptyset)$$

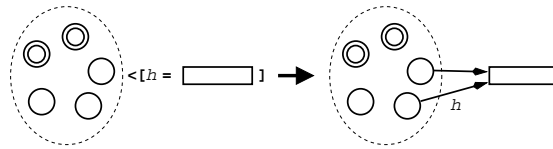
$$\text{gplug}(G_1, g, G_2, \tau) = (R_1, \\ E_1 \cup E_2 \cup \\ \{(n, g, m) \mid n \in \tau(g) \wedge m \in R_2\}, \\ \alpha_1 \sqcup \alpha_2)$$

$$\text{hplug}(G, h, s, \tau) = (R, E, \\ \lambda(n, h'). \text{if } n \in \tau(h) \text{ then } \alpha(n, h') \sqcup s \\ \text{else } \alpha(n, h'))$$

where $G_i = (R_i, E_i, \alpha_i)$ and $G = (R, E, \alpha)$. A careful inspection shows that all transfer functions are monotonic. The *frag* function constructs a tiny summary graph whose language contains only the given template. The *gplug* function joins the two summary graphs and adds edges from all relevant template gaps to the roots of the summary graph being inserted, which can be illustrated as follows:



The *hplug* function adds additional string values to the relevant attribute gaps:



We are now in a position to point out the need for the gap track analysis specified in Section 13.4. Without that initial analysis, the τ argument to *gplug* and *hplug* would always have to be the set \mathbf{N} of all constant template indices to maintain soundness. Plugging a value into a gap g would then be modeled by adding an edge from all nodes having a g gap, even from nodes that originate from completely unrelated parts of the source code or nodes where the g gaps already have been filled. For instance, it is likely that a program building lists as in the summary graph example in Section 13.4 would contain other templates with a gap named *items*. Requiring each gap name to appear only in one constant template would solve the problem, but such a restriction would limit the flexibility of the document construction mechanism significantly. Hence, we rely on a program analysis to disregard the irrelevant nodes when adding plug edges.

13.5.3 The Analysis

Since we are working with monotonic functions on finite lattices, we can again use standard iterative techniques to compute a least fixed point [174]. The proof of sound-

ness is omitted here, but it is similar to the one presented in [195].¹ The end result is for each program point ℓ an environment $summary_\ell : X \rightarrow G$ such that $L(summary_\ell(x_i))$ contains all possible XML templates that x_i may contain at ℓ . Those templates that are associated with show statements are required to validate with respect to the XHTML specification. We assume that the implicitly surrounding continue-button wrapper from Section 13.2 has been added already. Still, we must model the implicit plugging of empty templates and strings into the remaining gaps, so for the statement:

show x_i ;

with entry program point q , the summary graph that must validate with respect to the XHTML DTD is:

$$close(summary_\ell(x_i), track_\ell(x_i))$$

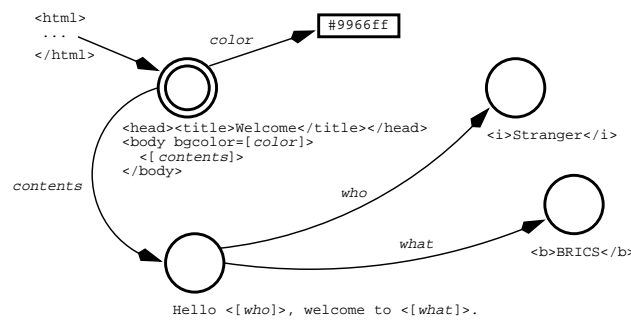
where $close$ is defined by:

$$close(G, \tau) = (R, \\ E \cup \{(n, g, m_\epsilon) \mid n \in \tau(g)\}, \\ \lambda(n, h). \text{if } n \in \tau(h) \text{ then } \alpha(n, h) \sqcup \{\epsilon\} \\ \text{else } \alpha(n, h))$$

where $G = (R, E, \alpha)$ and it is assumed that $\mathbf{f}(m_\epsilon) = \epsilon$. The $close$ function adds edges to an empty template for all remaining templates gaps, and adds the empty string as a possibility for all remaining attribute gaps.

13.5.4 The Example Revisited

For the small <bigwig> example in Section 13.2, the summary graph describing the document being shown to the client is inferred to be:



As expected for this simple case, the language of the summary graph contains exactly the single template actually being computed: Note that the XHTML template is implicitly completed with the <html> fragment.

¹The soundness of this algorithm relies on the assumption that all incoming branches to join points in the flow graph agree on which gaps are open. This is achieved using a simple preliminary program transformation that converts the “implicit ϵ -plugs” of <bigwig> [35] into explicit ones using the information from the DynDoc type system [195].

13.6 An Abstract DTD for XHTML

XHTML 1.0 is described by an official DTD [182]. We use a more abstract formalism which is in some ways more restrictive and in others strictly more expressive. In any case, the DTD for XHTML 1.0 can be captured along with some restrictions that merely appear as comments in the official version. We define an abstract DTD to be a quintuple:

$$D = (N, \rho, A, E, F)$$

where $N \subseteq \mathbf{E}$ is a set of *declared* element names, $\rho \in N$ is a *root* element name, $A : N \rightarrow 2^{\mathbf{A}}$ is an N -indexed family of attribute name declarations, $E : N \rightarrow 2^{N^\bullet}$ a family of element name declarations, and $F : N \rightarrow \Psi$ a family of formulas. We let $N^\bullet = N \cup \{\bullet\}$, where \bullet represents arbitrary character data.

Intuitively, an abstract DTD consists of a number of element declarations whereof one is designated as the root. Each element declaration consists of an element name, a set of allowed attribute names, a set of allowed contents, and a formula constraining the use of the element with respect to its attribute values and contents. A formula has the syntax:

$$\begin{aligned} \Psi &\rightarrow \Psi \wedge \Psi \\ &\rightarrow \Psi \vee \Psi \\ &\rightarrow \neg \Psi \\ &\rightarrow \text{true} \\ &\rightarrow \text{attr}(a) && a \in \mathbf{A} \\ &\rightarrow \text{content}(c) && c \in N^\bullet \\ &\rightarrow \text{order}(c_1, c_2) && c_i \in N^\bullet \\ &\rightarrow \text{value}(a, \{s_1, \dots, s_k\}) && a \in \mathbf{A}, k \geq 1, s_i \in \Sigma^* \end{aligned}$$

We define the *language* of D as follows:

$$L(D) = \{\rho(\delta)\varphi \mid D \models \rho(\delta)\varphi \wedge \text{gaps}(\varphi) = \emptyset\}$$

That is, the language is the set of documents where the root element is ρ and the acceptance relation \models is satisfied. This relation is defined inductively on templates as follows:

$$\begin{array}{c} \frac{}{D \models \varepsilon} \quad \frac{}{D \models \bullet} \\[10pt] \frac{D \models \varphi_1 \quad D \models \varphi_2}{D \models \varphi_1 \varphi_2} \\[10pt] \frac{\begin{array}{cc} \text{names}(\delta) \subseteq A(e) & D, \delta, \varphi \models F(e) \\ \text{set}(\varphi) \subseteq E(e) & D \models \varphi \end{array}}{D \models e(\delta)\varphi} \end{array}$$

For each element, it is checked that its attributes and contents are declared and that the associated formula is satisfied. The auxiliary functions *names* and *set* are formally defined by:

$$\begin{aligned} \text{names}(\varepsilon) &= \emptyset \\ \text{names}(a = s) &= \{a\} \\ \text{names}(a = h) &= \{a\} \\ \text{names}(\delta_1 \delta_2) &= \text{names}(\delta_1) \cup \text{names}(\delta_2) \end{aligned}$$

$$\begin{aligned}
set(\epsilon) &= \emptyset \\
set(\bullet) &= \{\bullet\} \\
set(g) &= \emptyset \\
set(e(\delta)\varphi) &= \{e\} \\
set(\varphi_1\varphi_2) &= set(\varphi_1) \cup set(\varphi_2)
\end{aligned}$$

On formulas, the \models relation is defined relative to the attributes and contents of an element:

$$\begin{aligned}
&\frac{D, \delta, \varphi \models \psi_1 \quad D, \delta, \varphi \models \psi_2}{D, \delta, \varphi \models \psi_1 \wedge \psi_2} \\
&\frac{D, \delta, \varphi \models \psi_1}{\varphi \models \psi_1 \vee \psi_2} \quad \frac{D, \delta, \varphi \models \psi_2}{\varphi \models \psi_1 \vee \psi_2} \\
&\frac{}{D, \delta, \varphi \models true} \quad \frac{D, \delta, \varphi \not\models \psi}{D, \delta, \varphi \models \neg \psi} \\
&\frac{a \in names(\delta)}{D, \delta, \varphi \models \mathbf{attr}(a)} \quad \frac{exists(word(\varphi), c)}{D, \delta, \varphi \models \mathbf{content}(c)} \\
&\frac{before(word(\varphi), c_1, c_2)}{D, \delta, \varphi \models \mathbf{order}(c_1, c_2)} \\
&\frac{a \notin names(\delta)}{D, \delta, \varphi \models \mathbf{value}(a, \{s_1, \dots, s_k\})} \\
&\frac{(a, s_i) \in atts(\delta) \quad 1 \leq i \leq k}{D, \delta, \varphi \models \mathbf{value}(a, \{s_1, \dots, s_k\})}
\end{aligned}$$

The $\mathbf{attr}(a)$ formula checks whether an attribute of name a is present, and $\mathbf{content}(c)$ checks whether c occurs in the contents. The $\mathbf{value}(a, \{s_1, \dots, s_k\})$ formula checks whether an a attribute has one of the values in s_1, \dots, s_k or is absent, and $\mathbf{order}(c_1, c_2)$ checks that no occurrence of c_1 comes after an occurrence of c_2 in the contents sequence. The auxiliary functions $atts$ and $word$ and the predicates $exists$ and $before$ are formally defined by:

$$\begin{aligned}
atts(\epsilon) &= \emptyset \\
atts(a = s) &= \{(a, s)\} \\
atts(a = h) &= \{(a, h)\} \\
atts(\delta_1\delta_2) &= atts(\delta_1) \cup atts(\delta_2) \\
word(\epsilon) &= \epsilon \\
word(\bullet) &= \bullet \\
word(g) &= \epsilon \\
word(e(\delta)\varphi) &= e \\
word(\varphi_1\varphi_2) &= word(\varphi_1)word(\varphi_2) \\
exists(w_1 \dots w_k, c) &\equiv \exists 1 \leq i \leq k : w_i = c \\
before(w_1 \dots w_k, c_1, c_2) &\equiv \forall 1 \leq i, j \leq k : \\
&\quad w_i = c_1 \wedge w_j = c_2 \Rightarrow i \leq j
\end{aligned}$$

Two common abbreviations are **unique**(c) \equiv **order**(c, c) (“ c occurs at most once”) and **exclude**(c_1, c_2) $\equiv \neg(\mathbf{content}(c_1) \wedge \mathbf{content}(c_2))$ (“ c_1 and c_2 exclude each other”).

Standard DTDs use restricted regular expressions to describe content sequences. Instead, we use boolean combinations of four basic predicates, each of which corresponds to a simple regular language. This is less expressive, since for example we cannot express that a content sequence must have exactly three occurrences of a given element. It is also, however, more expressive than DTDs since we allow the requirements on contents and attributes to be mixed in a formula. While the two formalism are thus theoretically incomparable, our experience is that XML languages described by DTDs or by more advanced schema languages typically are within the scope of our abstract notion.

13.6.1 Examples for XHTML

The DTD for XHTML 1.0 can easily be expressed in our formalism. The root element ρ is `html` and some examples of declarations and formulas are:

$$\begin{aligned} A(\text{html}) &= \{\text{xmlns}, \text{lang}, \text{xml:lang}, \text{dir}\} \\ E(\text{html}) &= \{\text{head}, \text{body}\} \\ F(\text{html}) &= \mathbf{value}(\text{dir}, \{\text{ltr}, \text{rtl}\}) \wedge \mathbf{content}(\text{head}) \wedge \\ &\quad \mathbf{content}(\text{body}) \wedge \mathbf{unique}(\text{head}) \wedge \\ &\quad \mathbf{unique}(\text{body}) \wedge \mathbf{order}(\text{head}, \text{body}) \\ \\ A(\text{head}) &= \{\text{lang}, \text{xml:lang}, \text{dir}, \text{profile}\} \\ E(\text{head}) &= \{\text{script}, \text{style}, \text{meta}, \text{link}, \text{object}, \text{isindex}, \\ &\quad \text{title}, \text{base}\} \\ F(\text{head}) &= \mathbf{value}(\text{dir}, \{\text{ltr}, \text{rtl}\}) \wedge \mathbf{content}(\text{title}) \wedge \\ &\quad \mathbf{unique}(\text{title}) \wedge \mathbf{unique}(\text{base}) \\ \\ A(\text{input}) &= \{\text{id}, \text{class}, \text{style}, \text{title}, \text{lang}, \text{xml:lang}, \\ &\quad \text{dir}, \text{onclick}, \text{ondblclick}, \text{onmousedown}, \\ &\quad \text{onmouseup}, \text{onmouseover}, \text{onmousemove}, \\ &\quad \text{onmouseout}, \text{onkeypress}, \text{onkeydown}, \\ &\quad \text{onkeyup}, \text{type}, \text{name}, \text{value}, \text{checked}, \\ &\quad \text{disabled}, \text{readonly}, \text{size}, \text{maxlength}, \\ &\quad \text{src}, \text{alt}, \text{usemap}, \text{tabindex}, \text{accesskey}, \\ &\quad \text{onfocus}, \text{onblur}, \text{onselect}, \text{onchange}, \\ &\quad \text{accept}, \text{align}\} \\ E(\text{input}) &= \emptyset \\ F(\text{input}) &= \mathbf{value}(\text{dir}, \{\text{ltr}, \text{rtl}\}) \wedge \\ &\quad \mathbf{value}(\text{checked}, \{\text{checked}\}) \wedge \\ &\quad \mathbf{value}(\text{disabled}, \{\text{disabled}\}) \wedge \\ &\quad \mathbf{value}(\text{readonly}, \{\text{readonly}\}) \wedge \\ &\quad \mathbf{value}(\text{align}, \{\text{top}, \text{middle}, \text{bottom}, \\ &\quad \quad \text{left}, \text{right}\}) \wedge \\ &\quad \mathbf{value}(\text{type}, \{\text{text}, \text{password}, \text{checkbox}, \\ &\quad \quad \text{radio}, \text{submit}, \text{reset}, \text{file}, \\ &\quad \quad \text{hidden}, \text{image}, \text{button}\}) \wedge \\ &\quad (\mathbf{value}(\text{type}, \{\text{submit}, \text{reset}\}) \vee \mathbf{attr}(\text{name})) \end{aligned}$$

In five instances we were able to express requirements that were only stated as comments in the official DTD, such as the last conjunct in $F(\text{input})$. The full description

of XHTML is available at <http://www.brics.dk/bigwig/xhtml/>.

13.6.2 Exceptions in <bigwig>

In one situation does <bigwig> allow non-standard XHTML notation. In the official DTD, the `ul` element is required to contain at least one `li` element. This is inconvenient, since the items of a list are often generated iteratively from a vector that may be empty. To facilitate this style of programming, <bigwig> allows empty `ul` elements but removes them at runtime before the XHTML is sent to the client. Accordingly, the abstract DTD that we employ differs from the official one in this respect. Similar exceptions are allowed for other kinds of lists and for tables. In the implementation, these fragment removal rules are specified the same way as the element constraints in the abstract DTD for XHTML, so essentially, we have just moved a few of the DTD constraints into a separate file.

13.7 Validating Summary Graphs

For every `show` statement, the data-flow analysis computes a summary graph $G = (R, E, \alpha)$. We must now for all such graphs decide the validation requirement:

$$L(G) \subseteq L(D)$$

for an abstract DTD $D = (N, \rho, A, E, F)$. The root element name requirement of D is first checked separately by verifying that:

$$\forall r \in R : \mathbf{f}(r) = \rho(\delta)\varphi \text{ for some } \delta \text{ and } \varphi$$

Then for each sub-template $e(\delta)\varphi$ of a template with index n in G we perform the following checks:

- $e \in N$ (the element is defined)
- $\text{names}(\delta) \subseteq A(e)$ (the attributes are declared)
- $\text{occurs}(n, \varphi) \subseteq E(e)$ (the content is declared)
- $n, \delta, \varphi \Vdash F(e)$ (the constraint is satisfied)

The validity relation \Vdash is given by:²

$$\frac{n, \delta, \varphi \Vdash \psi_1 \quad n, \delta, \varphi \Vdash \psi_2}{n, \delta, \varphi \Vdash \psi_1 \wedge \psi_2}$$

²Unfortunately, the rules for negation and disjunction presented here are flawed: The intended interpretation of $n, \delta, \varphi \Vdash \psi$ is that ψ is satisfied by *every* document represented by (n, δ, φ) . If, for instance, ψ_1 is satisfied by just one of those documents and ψ_2 by all other, then $\psi_1 \vee \psi_2$ will wrongfully evaluate to *false* and $\neg(\psi_1 \vee \psi_2)$ to *true*. The solution to this is as presented in [57] to introduce a four-valued logic with the truth value “*sometimes*” meaning that the formula is satisfied by some but not all documents represented by (n, δ, φ) , and “*don’t-know*” in case a precise answer cannot be provided, for instance by the disjunction of *sometimes* and *sometimes*. However, negations and disjunctions are in the abstract DTD for XHTML only used to express validity requirements beyond those definable by the standard DTD notation, so the flaws here do not impose a significant problem in practice.

$$\begin{array}{c}
\frac{n, \delta, \varphi \Vdash \psi_1}{n, \delta, \varphi \Vdash \psi_1 \vee \psi_2} \quad \frac{n, \delta, \varphi \Vdash \psi_2}{n, \delta, \varphi \Vdash \psi_1 \vee \psi_2} \\
\frac{}{n, \delta, \varphi \Vdash \text{true}} \quad \frac{n, \delta, \varphi \nVdash \psi}{n, \delta, \varphi \Vdash \neg \psi} \\
\frac{a \in \text{names}(\delta)}{n, \delta, \varphi \Vdash \text{attr}(a)} \quad \frac{c \in \text{occurs}(n, \varphi)}{n, \delta, \varphi \Vdash \text{content}(c)} \\
\frac{\text{order}(n, \varphi, c_1, c_2)}{n, \delta, \varphi \Vdash \text{order}(c_1, c_2)} \\
\frac{a \notin \text{names}(\delta)}{n, \delta, \varphi \Vdash \text{value}(a, \{s_1, \dots, s_k\})} \\
\frac{(a, s_i) \in \text{atts}(\delta) \quad 1 \leq i \leq k}{n, \delta, \varphi \Vdash \text{value}(a, \{s_1, \dots, s_k\})} \\
\frac{(a, h) \in \text{atts}(\delta) \quad \alpha(n, h) \subseteq \{s_1, \dots, s_k\}}{n, \delta, \varphi \Vdash \text{value}(a, \{s_1, \dots, s_k\})}
\end{array}$$

where *occurs* is the least function satisfying:

$$\begin{aligned}
\text{occurs}(n, \varepsilon) &= \emptyset \\
\text{occurs}(n, \bullet) &= \{\bullet\} \\
\text{occurs}(n, g) &= \bigcup_{(n, g, m) \in E} \text{occurs}(m, \mathbf{f}(m)) \\
\text{occurs}(n, e(\delta)\varphi) &= \{e\} \\
\text{occurs}(n, \varphi_1 \varphi_2) &= \text{occurs}(n, \varphi_1) \cup \text{occurs}(n, \varphi_2)
\end{aligned}$$

and *order* is the most restrictive function satisfying:

$$\begin{aligned}
\text{order}(n, \varepsilon, c_1, c_2) &= \text{true} \\
\text{order}(n, \bullet, c_1, c_2) &= \text{true} \\
\text{order}(n, g, c_1, c_2) &= \bigwedge_{(n, g, m) \in E} \text{order}(m, \mathbf{f}(m), c_1, c_2) \\
\text{order}(n, e(\delta)\varphi, c_1, c_2) &= \text{true} \\
\text{order}(n, \varphi_1 \varphi_2, c_1, c_2) &= \text{order}(n, \varphi_1, c_1, c_2) \wedge \\
&\quad \text{order}(n, \varphi_2, c_1, c_2) \wedge \\
&\quad \neg (c_2 \in \text{occurs}(n, \varphi_1) \wedge \\
&\quad \quad c_1 \in \text{occurs}(n, \varphi_2))
\end{aligned}$$

The definition of the validity relation is straightforward. It duals the definition of the acceptance relation in Section 13.6, except that we now have to take gaps into account. Only the auxiliary functions, *occurs* and *order*, are non-trivial. The function *occurs*(*n*, φ) finds the subset of N^\bullet that can occur as contents of the current element after plugging some gaps according to the summary graph, and *order*(*n*, φ , *c*₁, *c*₂) checks that it is not possible to obtain an *c*₂ before an *c*₁ in the contents φ . These two functions are defined as fixed points because the summary graphs may contain loops. In the implementation we ensure termination by applying memoization to the numerous calls to *occurs* and *order*.

Note that the validation algorithm is both sound and complete with respect to summary graphs: A graph is rejected if and only if its language contains a template that is

not in the language of the abstract DTD. Thus, in the whole validation analysis the only source of imprecision is the data-flow analysis that constructs the summary graph.³

Also note that our notion of abstract DTDs has a useful locality property: All requirements defined by an abstract DTD specify properties of single XML document nodes and their attributes and immediate contents, so if some requirement is not fulfilled by a given summary graph, it is possible to give a precise error message.

13.8 Experiments

The validation analysis has been fully implemented as part of the <bigwig> system using a monovariant data-flow analysis framework. It has then been applied to all available benchmarks, some of which are shown in the following table:

Name	Lines	Templates	Size	Shows	Time
chat	65	3	(0,5)	2	0.1
guess	75	6	(0,3)	6	0.1
calendar	77	5	(8,6)	2	0.1
xbiff	561	18	(4,12)	15	0.1
webboard	1,132	37	(34,18)	25	0.6
cdshop	1,709	36	(6,23)	25	0.5
jaoo	1,941	73	(49,14)	17	2.4
bachelor	2,535	137	(146,64)	15	8.2
courses	4,465	57	(50,45)	17	1.3
eatcs	5,345	133	(35,18)	114	6.7

The entries for each benchmark are its name, the lines of code derived from a pretty print of the source with all macros expanded, the number of templates, the size $(|E|, |\alpha|)$ of the largest summary graph, the number of show statements, and the analysis time in seconds (on an 800 MHz Pentium III with Linux).

The chat benchmark is a simple chat service, guess is a number guessing game, calendar shows a monthly calendar, xbiff is a soccer match reservation system, webboard is a bulletin board service, cdshop is a demonstration of an online shop, jaoo is a conference administration system, bachelor is a student management service, courses is a course administration system, and eatcs is a collection of services used by the EATCS organization. Some of the benchmarks are taken from the <bigwig> documentation, others are services currently being used or developed at BRICS.

The analysis found numerous validation errors in all benchmarks, which could then be fixed to yield flawless services. No false errors were reported. As seen in the table above, the enhanced compiler remains efficient and practical. The bachelor service constructs unusually complicated documents, which explains its high complexity.

³Because of the flaw mentioned on p. 214 the algorithm is in fact neither sound nor complete in general—however, for the particular abstract DTD for XHTML that we use, we have not encountered any wrong results to be produced in practice. As mentioned, these issues are definitively solved in the JWIG analyses [57] and a similar solution could be applied here.

13.8.1 Error Diagnostics

The <bigwig> compiler provides detailed diagnostic messages in case of validation errors. For the flawed example:

```

1 service {
2   html cover = <html>
3     <head><title>Welcome</title></head>
4     <body bgcolor=[color]>
5       <table><[contents]></table>
6     </body>
7   </html>;
8
9   html greeting = <html>
10    <td>Hello <[who]>,<br clear=[clear]>
11      welcome to <[what]>.
12    </td>
13  </html>;
14
15  html person = <html>
16    <i>Stranger</i>
17  </html>;
18
19  session welcome() {
20    html h;
21    h = cover<[color="#9966ff",
22      contents=greeting<[who=person],
23      clear="right"]>;
24    show h<[what=<html><b>BRICS</b></html>]>;
25  }
26 }
```

the compiler generates the following messages for the single show statement:

```

--- brics.wig:24: HTML validation:
brics.wig:4:
  warning: illegal attribute 'bgcolor' in 'body'
  template: <body bgcolor=[color]><form>...</form></body>

brics.wig:5:
  warning: possible illegal subelement 'td' of 'table'
  template: <table><[contents]></table>
  contents: td
  plugs: contents:{brics.wig:22}

brics.wig:10:
  warning: possible element constraint violation at 'br'
  template: <br clear=[clear]>/>
  constraint: value(clear,{left,all,right,clear,none})
  plugs: clear:{brics.wig:23}
```

At each error message, a line number of an XML element is printed together with an abbreviated form of the involved template, the names of the root elements of each template that can be plugged into the gaps, the constraint being violated, and the line numbers of the involved plug operations. Such reasonably precise error diagnostics is clearly useful for debugging.

13.9 Related Work

There are other languages for constructing XML documents that also consider validity. The XDuce language [110, 111] is a functional language in which XML templates are data types, with a constructor for each element name and pattern matching for deconstruction. A type is a regular expression over E^* . Type inference for pattern variables is supported. In comparison, we have a richer language and consequently need more expressive types that also describe the existence and capabilities of gaps. It seems unlikely that anything simpler than summary graphs would work. Also, we do not rely on type annotations. Since we perform an interprocedural data-flow analysis, we obtain a high degree of polymorphism that is difficult to express in a traditional type system. The XML language [155] compares similarly to our approach.

The initial design of the <bigwig> template mechanism was inspired by the MAWL language [144, 8]. The main difference is that MAWL only allows strings to be plugged into the gaps. Validating that MAWL programs only generate valid XHTML is therefore as easy as validating static documents, but such a simple document construction mechanism often becomes too restrictive for practical use. We have shown that using a highly flexible mechanism does not require validity guarantees to be sacrificed.

Most Web services are currently written either in Perl using CGI, in embedded scripting languages such as ASP, PHP, or JSP, or as server-integrated modules, for instance with Apache. Common to all these approaches is that there is no inherent type system for HTML or XML documents. In general, documents are constructed by concatenating text strings. These strings contain HTML or XML tags, attributes, etc., but the compiler or interpreter is completely unaware of that. This means that even *well-formedness*, that is, that tags are balanced and nested properly, which is one requirement for validity, becomes difficult to verify. We get that for free during parsing of the individual constant XML fragments and can concentrate on the many other validity requirements given by specific DTDs.

However, a common way of programming services in these languages is to use HTML or XML *constructor functions* to build documents more abstractly as trees instead of strings. This style is not enforced by the language, but if used consistently well-formedness is guaranteed. The difference between this and the <bigwig> style is that gaps in <bigwig> templates may appear non-locally, as described in Section 13.1, which gives a higher degree of flexibility. Since the constructor-based style is subsumed under the <bigwig> style as also described in Section 13.1, the summary graph technique could be applied for other languages.

13.10 Extensions and Future Work

Instead of our four basic predicates we could allow general regular expressions over the alphabet E^* . We could then still validate a summary graph, but this would reduce to deciding if a general context-free language is a subset of a regular language, which has an unwieldy algorithm compared to the simple transitive closures that we presently rely upon. Fortunately, our restricted regular languages appear sufficient. It is also possible to include many features from a richer XML schema language such as DSD [133], in particular context dependency and regular expression constraints on attribute values

and character data.⁴

Since our technique is parameterized in the choice of the abstract DTD, it easily generalizes to many other XML languages that can be described by such abstract DTDs. Finally, we could enrich `<bigwig>` with a set of operators for combining and deconstructing XML templates, making it a general XML transformation language.⁵ All such ideas readily permit analysis by means of summary graphs. However, a method for translating a DTD into a summary graph will be required.

13.11 Conclusion

We have combined a data-flow analysis with a generalized validation algorithm to enable the `<bigwig>` compiler to guarantee that all HTML or XHTML documents shown to the client are valid according to the official DTD. The analysis is efficient and does not generate many spurious error messages in practice. Furthermore, it provides precise error diagnostics in case a given program fails to verify.

Since our algorithm is parameterized with an abstract DTD, our technique generalizes in a straightforward manner to arbitrary XML languages that can be described by DTDs. In fact, we can even handle more expressive grammatical formalisms. The analysis has proved to be feasible for programs of realistic sizes. All this lends further support to the unique design of dynamic documents in the `<bigwig>` language.

⁴The program analyses in the **JWIG** system [57] are based on the notion of summary graphs introduced in this paper. In **JWIG**, we use general regular expressions for description of character data and attribute values, and we apply the DSD2 schema language for the validity analysis, as suggested here. Furthermore, we also apply summary graphs for checking the use of plug and receive operations which in `<bigwig>` is handled by the technique described in [195].

⁵Preliminary steps in this direction are made in [58] in the context of **JWIG**.

Chapter 14

The DSD Schema Language

with Nils Klarlund and Michael I. Schwartzbach

Abstract

XML (Extensible Markup Language), a linear syntax for trees, has gathered a remarkable amount of interest in industry. The acceptance of XML opens new venues for the application of formal methods such as specification of abstract syntax tree sets and tree transformations.

A user domain may be specified as a set of trees. For example, XHTML is a user domain corresponding to a set of XML documents that make sense as hypertext. A notation for defining such a set of XML trees is called a *schema language*. We believe that a useful schema notation must identify most of the syntactic requirements present in the user domains, and yet be sufficiently simple and easy to understand both by the schema authors and the users. Furthermore, it must allow efficient parsing and be modular and extensible to support reuse and evolution of descriptions.

In the present paper, we give a tutorial introduction to the DSD (Document Structure Description) notation as our bid on how to meet these requirements. The DSD notation was inspired by industrial needs. We show how DSDs help manage aspects of complex XML software through a case study about interactive voice response systems, i.e., automated telephone answering systems, where input is through the telephone keypad or speech recognition.

The expressiveness of DSDs goes beyond the DTD schema concept that is already part of XML. We advocate the use of nonterminals in a top-down manner, coupled with boolean logic and regular expressions to describe how constraints on tree nodes depend on their context. We also support a general, declarative mechanism for inserting default elements and attributes. Also, we include a simple technique for reusing and evolving DSDs through selective redefinitions. The expressiveness of DSD is comparable to that of the schema language XML Schema proposed by W3C, but their syntactic and semantic definition is significantly larger and more complex. Also, the DSD notation is self-describable: the syntax of legal DSD documents including all static semantic requirements can be expressed within the DSD language itself.

14.1 Introduction

XML (Extensible Markup Language) [45] is a syntax derived from SGML for markup of text. XML is particularly interesting to computer scientists because the markup notation is really nothing but a way of specifying labeled trees. The tree view and the convenient SGML syntax of HTML have been important to the development of the World Wide Web. Thus, it may not be surprising that XML syntax since its introduction in 1998 has been hyped as a universal solution to the pervasive problem of format incompatibility.

Such generous promises notwithstanding, at least one fascinating and fundamental quality sets XML-based notations apart from ad hoc syntax: they encourage tree transformations—a technique that application programmers usually do not take advantage of. In fact, it would probably be considered a hassle even to define a set of parse trees and procedures according to which they are constructed and parsed. XML circumvents this problem by offering a primary representation based on trees, at the expense of syntactic succinctness. Of course, trees and mappings between trees are a main ingredient of computer science. For example, such mappings are essential to building compilers, where the compilation process is partitioned into several phases, most of which simply transform one intermediate tree format into another one. XML has been suggested as an underlying notation for structuring and manipulating information in general. As a foundation of this, XML schemas are needed to formalize the sets of parse trees that constitute the individual languages.

The purpose of the present article is to indicate how XML opens new ways of applying formal computer science techniques to general, practical problems. Specifically, we study the formal specification of XML languages, that is, sets of abstract syntax trees, and default insertion mechanisms for common tree transformations needed by application programmers. Both aspects are part of the DSD (Document Structure Description) notation, which we introduce informally in this article. Before we explain DSDs, let us mention some fundamental XML technologies that are already standardized (in the sense of being a W3C Recommendation) or under development:

- *Syntax*: Schemas describe the formal syntax of XML languages. As for other formal languages, a precise syntax description provides an essential basis, both for the tool builders and the application users. XML has inherited the DTD schema concept from SGML, but this notation is considered inadequate by many. The newest schema notation from W3C is called XML Schema [209] and it has recently achieved Recommendation status. However, as explained later, this language is in our opinion not satisfactory, and several alternatives have been proposed.
- *Transformation*: Since XML encourages construction of highly specialized languages, there is a strong need for domain-specific languages that allow general transformations between XML languages to be defined more easily than possible with general-purpose programming languages. XSLT [59], the transformation part of the XSL language, became an official recommendation in 1999 and has become very popular.
- *Style sheets*: CSS (Cascading Style Sheets) is an example of a specialized trans-

formation language, designed to make visual rendering for XML (and HTML) documents, which is a typical kind of transformation. It consists of a simple tree transformation language and a target language of text properties for layout. CSS2 [30] is the latest official recommendation.

- *Database querying*: Since XML documents in a sense generalize the relational database model to general semi-structured, there is a need for corresponding generalizations of query languages. A draft specification of the XQuery language [27] has recently been published. The term *schema* originates from the database community where it denotes descriptions of the structure of relations.
- *Linking and addressing*: XML is designed to operate on the Web, so notations for defining links between documents and for addressing fragments of documents are essential. XLink [73] allows generalized links between XML resources to be defined. It is based on XPointer [72], which in turn uses XPath [61] for expressing locations in XML documents in a robust manner. XPath is also used in XML Schema to express uniqueness constraints, in XSLT as a pattern matching mechanism, and in XQuery to express basic queries.
- *Namespaces*: XML languages are often built on top of other XML languages. This introduces the demand for a name space mechanism to be able to distinguish the various parts of an XML document. XML Namespaces [44] allows URIs to be associated with XML markup to be able to uniquely determine which sublanguages the markup belongs to. We mention namespaces here because they have implications to essentially all other XML technologies, in particular schema languages.

For a more thorough introduction to these concepts, we refer to [162]. Agreeing on well-designed languages for these fundamental technologies allows generic tools to solve problems common to many XML application languages. Agreeing on a simple but powerful schema language has the additional benefit of making it easier to design and learn new XML languages. In the area of programming languages, the BNF notation is an example of this phenomenon. Unarguably, the simplicity of that notation has been a requisite for its widespread use.

In the area of schema languages, numerous proposals, such as DDML [32], DCD [43], XML-Data [145], XDR [92], SOX [70], TREX [60], Schematron [119], Assertion Grammars [184], and RELAX [169], have already emerged. Recently, W3C has issued their XML Schema proposal [209] in an attempt to reconcile the efforts. However, it has been met with intense debate, primarily due to its unprecedented complexity viewed by many as being unnecessary and harmful [191, 2]. Concurrently, RELAX NG [62] has been developed as a descendant of RELAX and TREX and is now being standardized by OASIS. The many proposals, and the outcome of the XML Schema effort, indicate that it is far from obvious how the right schema language should be designed. In general, the XML notation turns out to be so versatile that it is hard to satisfy all design requirements and capture the various usage patterns, and at the same time keep the schema notation simple. We give a more thorough comparison between DSD and the most significant alternatives in Section 14.7.

Our DSD proposal—which is rigorously defined in [132]—has the ambition of providing an expressive power comparable to that of XML Schema and RELAX NG,

while at the same time remaining simple to learn and use. We have tried to identify the most central aspects of XML language syntax and turn these into a clean set of schema constructs, based on well-known computer science concepts, such as boolean logic and regular expressions. A DSD defines a grammar for a class of XML documents, including documentation for that class, and additionally a CSS-like notation for specifying default parts of documents. As most other schema language proposals, the DSD language itself uses the XML notation. This opens up for the possibility of being self-describing, that is, having a DSD description of the DSD language.

We recall that an XML document consists of named *elements* representing tree nodes. Elements have *attributes*, representing name/value pairs, and *content*, which is Unicode text called *chardata*, interspersed with subelements. We here ignore comments and DTD information, and we assume that entity references have been expanded. For example, consider the following XHTML document fragment:

```
<body class='mystuff'>
  Hello <em>there</em>
</body>
```

This fragment contains an element named `body` that corresponds to a tree node labeled `body`. The node has an attribute named `class` and two children corresponding to its content, that is, the part between the start tag `<body . . .>` and the end tag `</body>`. The first child is a text node with value `Hello`, and the other is an element node labeled `em`. The `em` node in turn has one child node, which is a text node. The markup is required to be *well-formed*, meaning that the begin and end tags are balanced and nested properly, which allows us to view XML documents as tree structures. A schema for XHTML would for example state that `class` attributes in fact are allowed in `body` elements, that `chardata` is allowed in the content, but also that for instance `body` elements cannot appear within the `em` tags. A schema language should make it possible to easily express such constraints.

Besides basing the DSD design on simple concepts that are familiar to computer scientists, we have a number of more technical goals for the descriptive power of the DSD notation. These goals are by no means comprehensive, but they reflect most of the needs we have seen in document processing and database applications:

- DSD should allow context dependent descriptions of content and attributes, since the context of a node, such as ancestors and attribute values, often governs what is legal syntax.
- Default attribute values and content should be defined in a declarative manner, separate from the structural descriptions. Thus we seek a generalization of CSS so that defaultable properties in the form of attributes and element content can be defined for arbitrary XML domains. CSS manipulates defaults, but only for properties in predefined formatting models.
- As most other schema languages, DSD should support node IDs and references for expressing non-tree-structured data. In addition, it should permit the description of what references may point to.
- In order to support development and maintenance of large schemas, DSD should contain mechanisms for schema evolution and reuse.

- DSD should be self-describable. This property allows schemas themselves to be viewed as application documents.
- The content model should be flexible enough to allow ordered and unordered content to be mixed.
- It should be possible to intersperse informal documentation with the formal language of schemas. That allows them to serve as complete language descriptions.
- Validity of chardata and attribute values should be defined with an extensible mechanism so that only a minimal number of primitive types are included in the core language.
- DSD should complement XSLT in the sense that assumptions made by XSLT style sheets about the shape of input documents can be made explicit.
- Finally, it is also important to us that a DSD yields a linear-time algorithm for checking conformance of XML documents.

To honor these ambitions, our design combines several elementary ideas: a uniform notion of *constraint* that captures the legality of attributes, attribute values, and content; *conditional constraints* guarded by *boolean expressions* that capture dependencies between attributes, attribute values, element contexts, and content; *nonterminals* in the form of element IDs that allow several different versions of an element to coexist; the concept of *projected content* that allows succinct descriptions of both ordered and unordered content; *regular expressions* to describe both attribute values and content sequences; automatic insertion of *default* attributes and element content guided by boolean expressions; a simple notion of redefinition combined with a schema inclusion mechanism for supporting extension and modularity; and *points-to* requirements that constrain the targets of references.

The only major omission is the concept of namespaces, whose semantics until recently has been the subject of controversy [31]. In the current version of DSD, we do apply namespaces within the DSD language, but we do not support namespaces in the application languages. We plan to add proper support for namespaces in a future version to mend this limitation.

Naturally, there are constraints that within reason can be conceived but are not expressible in our formalism. Moving to Turing complete formalisms would complicate the language unnecessarily. As in programming language grammar formalisms, it is customary to supplement a grammatical check with a few specialized routines written in a general programming language.

Despite its expressive power, the DSD language is simple enough that it can be rigorously defined in 15 pages [132] (where the page count excludes examples and introduction). The specification of the Structural Part of XML Schema runs to about 140 pages (counted in the same way). The present paper describes the main ideas of the DSD notation and relates it to other XML schema language proposals. We also provide an account of an industrial example that motivated DSD: HTML-like languages for defining Interactive Voice Response (IVR) systems, which are user interfaces that work through spoken prompts and telephone pad or speech input.

The main contribution of this work is the attempt to simplify and yet generalize existing XML schema languages. Also, we believe to have identified some essential design requirements and show that in particular boolean logic and regular expressions are useful formalisms in schema languages.

14.1.1 Outline

After an overview of the XML tree model in Section 14.2, we introduce the DSD concepts through little examples in Section 14.3, and we explain the notion of a meta-DSD. In Section 14.4, we present a complete DSD example for information about books. In Section 14.5, we describe a prototype implementation of the DSD processor, and in Section 14.6, we discuss how an application programmer would benefit from DSDs when learning and using a domain specific language for IVR applications. In Section 14.7, we discuss related work, in particular XML Schema and RELAX NG. We conclude in Section 14.8 with a summary of our experiences with DSD, followed by plans and ideas for future development.

14.2 XML Concepts

The reader is assumed familiar with the most common XML concepts (XML is officially defined in [45]). However, since there unfortunately is no common agreement on the terminology, we now give a brief description of the XML data model used in DSD.

A well-formed XML document is represented as a tree. The leaves correspond to empty elements, chardata, processing instructions, and comments. The internal nodes correspond to non-empty elements. For that reason, we often confound the terms “element” and “node”. DTD information is not represented in the tree. Each element is labeled with a name and a set of attributes, which each consists of a name and a value. Names, values, and chardata are Unicode strings [206].

Child nodes are ordered. The *content* of an element is the sequence of its immediate child nodes. The *context* of a node is the path of nodes from the root of the tree to the node itself. Element nodes are ordered according to *document order*: an element *a* is *before* an element *b* if the start tag of *a* occurs before the start tag of *b* in the usual textual representation of the XML tree. We will assume that trees are a normalized by a process that combines adjacent text nodes by concatenating their text.

Processing instructions with target `dsd` or `include`, as well as elements and attributes with namespace `http://www.brics.dk/DSD`, contain information relevant to the DSD processing. All other processing instructions and also chardata consisting of white-space only and comments are ignored.

14.3 The DSD Language

A DSD defines the syntax of a family of conforming XML documents. An *application document* is an XML document intended to conform to a given DSD. It is the job of a *DSD processor* to determine whether or not an application document is conforming. A

DSD is itself an XML document. This section describes the main aspects of the DSD language and its meaning. For a complete definition, we refer to [132].

A DSD can be associated with an application document by placing a special processing instruction in the document prolog. This processing instruction has the form

```
<?dsd URI="URI"?>
```

where *URI* is the location of the DSD. By inserting this in the application document, the author states that the document is intended to conform to the designated DSD.

A DSD processor basically performs one top-down traversal of the application document tree in order to check conformance. During this traversal, constraints and other requirements from the DSD are *evaluated* relative to a *current element* of the application document. The DSD processor consults the DSD to determine the constraints that are *assigned* to each node for later evaluation. Initially, a constraint is assigned to the root node. Evaluation of a constraint may entail the insertion of default attribute values and default content in the current element. Also, it may assign constraints to the subelements of the current element. If no constraints are violated during the entire tree traversal, the original document conforms to the DSD. The document augmented with inserted defaults constitutes the result of the DSD processing.

A DSD consists of a number of definitions, each associated with an ID for reference. In the following, the various kinds of DSD definitions are described. We use a number of small examples, some inspired by the XHTML language [182] and some that are fragments of the book example described in Section 14.4.

14.3.1 Element Constraints

The central definition in DSD is the *element definition*. An element definition specifies an element name and a *constraint*. During conformance checking, each element node in the application document is assigned an ID referring to an element definition from the DSD. In order for the element node to match the element definition, they must have the same name, and the element node must satisfy the constraint.

The IDs of element definitions are reminiscent of nonterminals in context-free grammars. Each ID determines the syntactic requirements imposed on the content, attributes, and context of the elements to which it is assigned. We distinguish between definition IDs and element names in order to allow several versions of an element to coexist. Thus, several different element definitions may occur with the same name. To avoid confusion about the term “ID”, note that element definition IDs are references into the DSD and that multiple application document elements may be assigned the same ID.

As an example, consider a DSD describing a simple database containing information about books, such as, their titles, authors, ISBN numbers, and so on. Imagine that both the whole database and each book entry must contain a `title` element, but with different structures. Book entry titles may contain only chardata and no markup, and defaults may be specified for them. Database titles may on the other hand contain arbitrary content and no attributes, and cannot be given by defaults. These two kinds of `title` elements can be defined as follows:

```

<ElementDef ID="book-title" Name="title" Defaultable="yes">
  <Content><StringType/></Content>
</ElementDef>

<ElementDef ID="database-title" Name="title">
  <ZeroOrMore><Union>
    <StringType/><AnyElement/>
  </Union></ZeroOrMore>
</ElementDef>

```

A constraint is defined by a number of constraint expressions, which can contain declarations of attributes and element content, boolean expressions about attributes and context, and conditional subconstraints guarded by boolean expressions. The constraint is satisfied if the evaluation of each constituent succeeds. These aspects are described in the following sections.

The example below expresses something that is impossible or cumbersome to formalize in other schema proposals, namely the requirement that anchor elements in XHTML are not nested:

```

<ElementDef ID="a">
  <Constraint><Not><Context>
    <Element Name="a"/><SomeElements/>
  </Context></Not></Constraint>
</ElementDef>

```

This element definition contains a single constraint expression, which is a simple boolean expression querying the element context. Note that the name attribute of the `ElementDef` is missing here. That simply means that the name is the same as the ID.

In DTD, the anchor nesting restriction cannot be formalized and merely appears as a comment. The DTD does exclude a elements from appearing immediately below other a elements, but, for instance, it allows `<a><a>...`. Most other schema languages, including XML Schema, has the same limitation.

Boolean expressions are built from the usual boolean operators, `And`, `Or`, `Not`, `Imply`, etc., and are used for several purposes: they express dependencies between attributes, and they are used as guards in conditional constraints and default declarations, as explained later.

14.3.2 Attribute Declarations

During evaluation of a constraint, attributes are *declared* gradually. Only attributes that have been declared are allowed in an element. Since constraints can be conditional and attributes are declared inside constraints, this evaluation scheme allows hierarchical structures of attributes to be defined. Such structures cannot be described by other schema proposals although they are common. For instance, in an XHTML input element, the `length` attribute may be present only if the `type` attribute is present and has value `text` or `password`. In most schema languages, this kind of constrain is not expressible. Their solution is to allow all combinations and resort to other means, typically general programming languages, for expressing the extra requirements. However, since dependencies are a very common phenomenon in XML languages, this is clearly

not satisfactory. Another typical example can be found in the XML Schema specification [209], Section 3.2.3: “default and fixed may not both be present [...] if default and use are both present, use must have the actual value optional [...] if ref is present, then all of <simpleType>, form and type must be absent”. Surprisingly, even though the XML Schema language repeatedly uses such dependencies itself, they cannot be expressed in XML Schema. In contrast, the conditional constraints and boolean expressions in DSD capture this notion of dependencies in a straightforward manner.

An *attribute declaration* consists of a name and a string type. The name specifies the name of the attribute, and the string type specifies the set of its allowed values. Unless it is declared as optional, an attribute must be present if it is declared. Conversely, only declared attributes are allowed to be present.

The presence and values of declared attributes can be tested in boolean expressions and context patterns. For instance, the expression:

```

<Attribute name="action">
  <StringType IDRef="URI"/>
</Attribute>
```

evaluates to *true* if and only if the attribute named `action` satisfies two conditions: it has been declared and it is present in the current element with a value matching the string type `URI`.

The CSS language can assign properties to the elements in a document, based on context-sensitive selectors. In generic XML settings where properties appear as element attributes, such as in SMIL [109], this can lead to semantic ambiguities since setting and testing of attributes occurs in no pre-defined order. Our notion of gradual attribute declaration avoids such ambiguities.

14.3.3 String Types

A *string type* is a set of strings defined by a regular expression. String types are used for two purposes: to define valid attribute values and to define valid chardata.

Regular expressions provide a simple, well-known, and expressive formalism for specification of sets of strings. Many reasonable sets can be defined, and by the correspondence with finite-state automata, an efficient implementation is possible. A rich set of operators is provided, such as *Sequence*, *ZeroOrMore*, *Union*, *Optional*, *Intersection*, and *Complement*.

The use of regular expressions is more flexible than using a predefined collection of data types. Special automata representations for large alphabets hold the promise that the efficient regular expression implementations extend to Unicode.¹

Most well-known data types, such as URIs, email addresses, and ZIP codes, can be described by regular expressions. The following example shows the definition of ISBN numbers:

```

<StringTypeDef ID="isbn">
  <Sequence>
    <Repeat Value="9">
      <Sequence>
```

¹See e.g. <http://www.brics.dk/automaton/>

```

        <CharRange Start="0" End="9"/>
        <Optional><CharSet Value=" -"/></Optional>
    </Sequence>
</Repeat>
<CharSet Value="0123456789X"/>
</Sequence>
</StringTypeDef>

```

This defines ISBN numbers to consist of 10 digits, optionally separated by single blanks or dashes, and where the final digit may also be the character 'X'. In a more familiar notation, this regular expression would be written as $([0-9] \text{ } [-]?)\{9\}[0-9X]$. The benefit of our more voluminous notation is that the syntactic structure of the expression is immediate from the XML structure.

In comparison, other schema languages typically provide a number of predefined data types and focus less on flexibility and user defined types. More details are given in Section 14.7.

14.3.4 Content Expressions

Recall that the content of an element is a sequence of element nodes and chardata nodes. *Content expressions* are used to specify sets of such sequences. These expressions are a kind of regular expression that occur in element constraints.

Content expressions are built of atomic expressions and content expression operators. An atomic expression is either an element description or a string type. An element description is essentially a reference to an element definition. It matches a given element node if their names match. The string types specify chardata child nodes. Checking that content sequences satisfy the given constraints has the side-effect that element definition IDs are assigned to the subelements. Also, as explained in Section 14.3.6, insertion of default content occurs while checking content expressions. Because of these side-effects, we need a non-standard interpretation of the regular expression constructs occurring in content expressions, in order to get a well-defined behavior.

The content expression operators include *Sequence*, *ZeroOrMore*, *AnyElement*, *Union* and *If*. A *Sequence* is matched with a content expression by a left-to-right traversal. For *ZeroOrMore*, the traversal is eager, that is, it continues as long as there is a match of the subexpression. For *Union*, the traversal allows backtracking. Each option is tried, and the first one that matches is chosen. The *If* construct defines a conditional subexpression.

As an example, the valid content of a XHTML table element (see [182], App. A.1) can be described by the following content expression:

```

<Sequence>
  <Optional><Element IDRef="caption"/></Optional>
  <Union>
    <ZeroOrMore><Element IDRef="thead"/></ZeroOrMore>
    <ZeroOrMore><Element IDRef="tfoot"/></ZeroOrMore>
  </Union>
  <Optional><Element IDRef="thead"/></Optional>
  <Optional><Element IDRef="tfoot"/></Optional>

```

```

<Union>
  <OneOrMore><Element IDRef="tbody"/></OneOrMore>
  <OneOrMore><Element IDRef="tr"/></OneOrMore>
</Union>
</Sequence>

```

Ignoring the syntactic overhead of the XML notation, this example could just as easily be expressed in DTD. But, as explained in the following, DSDs also allow more complex content requirements to be specified.

A constraint may contain more than one content expression. Each of them then must match some of the content of the current element, just like each attribute declaration must match an attribute. More precisely, each content expression is matched against a subsequence of the content that consists of elements mentioned in the content expression itself. Thus, the actual content is *projected* onto the elements that the content expression contains. If, for instance, a content expression mentions elements A and B, and the content is a sequence of elements A, B, C, followed by a chardata node and an element A, then this expression is matched against the projected content A, B, A. This method makes it easy to combine requirements of both *ordered* and *unordered* content. Additionally, unordered content is declared just like attributes.

In the XHTML specification, the content of the head element is described as “head.misc, combined with a single title and an optional base element in any order”. In a DTD, this requirement can be formalized only by listing all the possible combinations in a single regular expression. The XML Schema proposal introduces a separate operator to express interleavings, however, it cannot be combined arbitrarily with the other content description operators. With DSD, a simple constraint with three content expressions does the job:

```

<Content IDRef="head.misc"/>
<Element IDRef="title"/>
<Optional><Element IDRef="base"/></Optional>

```

When such a set of content expressions is evaluated, each of them is evaluated on the *projected content*, namely the subsequence of the content that mentions the element names in the expression. The first expression only looks at the elements that occur in head.misc (which is defined elsewhere); the second only looks at title elements and states that there must be exactly one of these; and the third expression states that there can be an optional base element somewhere in the content. Additionally, each content node must be matched by exactly one content expression. Thus, generally speaking, content expressions in a constraint must not overlap with respect to element names they mention, just as it is an error to declare an attribute more than once. This simple and intuitive approach is unique to DSD.

For another example, consider the combination of the following two content expressions:

```

<Sequence>
  <Element IDRef="first"/>
  <Element IDRef="initial"/>
  <Element IDRef="last"/>

```

```
</Sequence>
<Optional><Element IDRef="homepage"/></Optional>
```

Together they require that the content consists of the three elements *first*, *initial*, and *last* occurring in that order, and that a single *homepage* element may optionally occur anywhere in that sequence. Without multiple content expressions and the notion of projected content, all possible combinations would have to be explicitly listed.

As explained in Section 14.7, the content description mechanisms in other schema languages are typically also based on variations of regular expressions, in some cases adding a notion of inheritance. The solution to the problem of expressing combined ordered and unordered content is however unique to DSD.

14.3.5 Context Patterns

A *context pattern* can be used with defaults, constraints and content descriptions to make them context dependent.

Context patterns are very similar to CSS selectors [30]. A context pattern is a sequence of context terms. A *context term* is either an element pattern or a *SomeElements* element. An *element pattern* specifies an element name and a set of attributes. Recall that we define the *context* of the current element to be the sequence of nodes that start at the root of the XML tree and end in the current element.

Before summarizing the meaning of context patterns, we provide an example of a context pattern that matches those *li* elements immediately within *ul* elements inside *form* elements whose *method* attribute has value *post*:

```
<Context>
  <Element Name="form">
    <Attribute Name="method" Value="post"/>
  </Element>
  <SomeElements/>
  <Element Name="ul"/>
  <Element Name="li"/>
</Context>
```

The matching semantics of contexts is as follows. The context of the current element is matched by a context pattern if the context can be decomposed into consecutive fragments such that the sequence of fragments matches the sequence of context terms in the pattern. An element pattern matches a single element node if the name and attributes match. A *SomeElements* matches any context fragment. Implicitly, all context patterns begin with a *SomeElements* element.

To see how useful context-dependent definitions are, let us consider a common situation: an XML grammar that represents not one but several related XML notations. For example, a DSD may specify both draft and final markup notations for books. This is the scenario mentioned in the XML 1.0 specification, where conditional sections of DTDs may be used to describe variations:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >
<![%draft;[
```



```

<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>

```

Here, two flags (parameter entities), called `draft` and `final`, are used to control the expansion of the two conditional definitions of `book`. Typically, these flags would be declared in the document type declaration of the application document, whereas the conditional sections would be declared in an external DTD. The declarations in the application document are processed before the external DTD.

As stated, the first conditional definition is expanded since the first item of the conditional definition expands to `INCLUDE`. Similarly, the second definition is not expanded since the first item expands to `IGNORE`. This mechanism is somewhat unsafe: a document writer must set two flags at the same time, and their values must be opposite each other.

With DSDs, the parameterization of the XML grammar can be explained in terms of the application document itself. For example, if the root element is called `DOC`, then an attribute `draft` of this element would govern the definition of a book:

```

<ElementDef ID="book">
  <Sequence>
    <If>
      <Context>
        <Element Name="DOC">
          <Attribute Name="draft" Value="true"/>
        </Element><SomeElements/>
      </Context>
      <Then><ZeroOrMore>
        <Element IDRef="comments"/>
      </ZeroOrMore></Then>
    </If>
    <Element IDRef="title"/>
    <Element IDRef="body"/>
    <Optional><Element IDRef="supplements"/></Optional>
  </Sequence>
</ElementDef>

```

Here the logic of the different versions is clearly spelled out at the XML level of the application document itself. We believe that expressing this logic is not possible with any of the other schema language proposals, since they do not have equivalent notions of context expressions and conditional constraints. One exception is Schematron, which employs the powerful language XPath for expressing constraints, as explained in Section 14.7.

14.3.6 Default Insertion

It is convenient to application document authors to be able to omit implied attributes and other document parts. Since schemas describe the document structure, they are a suitable place to specify default values for such parts. Validating a document then has the side-effect of inserting the defaults, which is often useful to subsequent document processing.

In DSD, default attributes and content are defined by an association to a boolean expression. Such attributes or content is *applicable* for insertion at a given place in the application document if the boolean expression evaluates to true at that place.

In other schema languages, the most common approach is to specify the defaults together with the structural descriptions, for instance at the attribute declarations. However, by specifying the defaults separately in a declarative manner, the default mechanism becomes more flexible because it allows variations of the default values. The IVR application shown in Section 14.6 utilizes this property extensively.

The following example defines that the length of input fields of type text is by default 20:

```
<Default>
  <Context>
    <Element Name="input">
      <Attribute Name="type" Value="text"/>
    </Element>
  </Context>
  <DefaultAttribute Name="length" Value="20"/>
</Default>
```

Defaults are inserted “upon request” by constraints:

- When an attribute declaration is encountered and the declared attribute is not present in the current element, an applicable default is inserted, if any exists.
- During evaluation of a content expression, if an element description or a string type is encountered and the next content node does not match the description, then an applicable default is inserted, if any exists. Default elements can be inserted only if declared as defaultable by the description.

A notion of *specificity* of defaults, reminiscent of CSS [30], is used to determine a default when more than one is applicable. Intuitively, the default with the most complex boolean expression is chosen. If two are equally complex, the one latest defined is chosen.

For convenience, defaults can also be defined in the application document. Every application document element may contain default definitions, which in a sense extend the DSD. Such default definitions are recognized using the DSD namespace. They are not considered part of the application document by the DSD processor. Their scope is not the whole application document. Instead, they are considered as applicable only to the subtree rooted by the element in which they occur.

The following example shows how the length default previously defined may be overridden for certain text type input elements, namely those inside form elements that have an action attribute whose value is a string starting with the prefix `http://www.brics.dk/`:

```
<DSD:Default>
  <Context>
    <Element Name="form">
      <Attribute Name="action"/>
      <Sequence>
        <String Value="http://www.brics.dk/" />
        <ZeroOrMore><AnyChar/></ZeroOrMore>
      </Sequence>
    </Element>
    <SomeElements/>
    <Element Name="input">
      <Attribute Name="type" Value="text"/>
    </Element>
  </Context>
  <DefaultAttribute Name="length" Value="30"/>
</DSD:Default>
```

Analogously to CSS, defaults defined in the application document are always considered more specific than defaults defined in the DSD document. Moreover, when two application document defaults are applicable and they are not siblings, the one with the smallest scope, that is, the innermost one, will always be considered more specific than the other.

Most other schema languages contain a default mechanism. However, some only support attribute defaults or content defaults that only contain chardata. Only DSD allows individual elements to be inserted in content sequences, and it is also unique in separating the default declarations from the structural descriptions. In Section 14.6, we will look at examples that involve managing a great number of interdependent defaults.

14.3.7 ID Attributes and Points-To Requirements

In attribute declarations, a DSD may declare that application document attributes are of type ID or IDRef, as also possible with DTDs. An attribute of type ID is considered a *definition* of the value of the attribute. Such a definition must be unique. Similarly, an IDRef attribute is a *reference* to the element containing the attribute defining the given value, and such an element must exist.

Additionally, a DSD may impose a *points-to* requirement on the element denoted by a reference. Such a requirement is defined by a boolean expression, which may probe attribute values and context as we have seen. This unique mechanism allows a more precise description of semi-structured data. An example is the DSD notation itself, as shown in Section 14.3.10.

In the following example, a book-reference attribute is declared. It must refer to an element with an attribute of type ID occurring in a book element:

```
<AttributeDecl ID="book-reference" IDType="IDRef">
  <PointsTo>
    <Context><Element Name="book"/></Context>
  </PointsTo>
</AttributeDecl>
```

The ID definitions, IDRef references, and points-to requirements are checked in a separate phase after the main traversal of the application document.

14.3.8 Redefinitions and Evolving DSDs

Many XML languages are built from existing languages. Also, often a whole family of related languages is to be defined. DSDs support these software practices by providing two simple mechanisms: *document inclusion* and *redefinition*. This allows schemas to be created from existing schemas through modifications and extensions.

Both DSD documents and application documents can be created as extensions of other documents using a special `include` processing instruction of the form:

```
<?include URI="URI"?>
```

where *URI* denotes the document to be included, that is, inserted in place of the processing instruction. A document can only be included once into a given document; subsequent attempts are ignored.

In DSDs, all definitions can be renewed. One can include a document containing a definition of a concept and then later redefine the concept. Since the DSD language is designed to be self-describable, the meta-DSD must be able to express this notion of redefinition.

To accommodate modifications of DSD definitions, two new attribute types, `RenewID` and `CurrIDRef`, are introduced beside `ID` and `IDRef`. All definitions can be redefined using `RenewID`. An `IDRef` attribute refers to the *final* definition or redefinition in the document for that ID. An attribute of type `CurrIDRef` refers to the *current definition*, which is the last definition or redefinition occurring before the reference and that does not contain it. Assume that in some existing DSD a book element has been defined as follows:

```
<ElementDef ID="book">
  <Constraint IDRef="book-constraints"/>
</ElementDef>

<ConstraintDef ID="book-constraints">
  ...
</ConstraintDef>
```

Consider a situation where we want to reuse this DSD but would like to extend the book constraints with a new attribute declaration. This can be done using `RenewID` to redefine `book-constraint` and `CurrIDRef` to refer to the original definition:

```
<ConstraintDef RenewID="book-constraints">  
  <Constraint CurrIDRef="book-constraints"/>  
  <AttributeDecl Name="new-attribute"/>  
</ConstraintDef>
```

Most schema languages support modularization using `include`-like features. Some also allow redefinitions, but without being able to refer to the old definitions as our `CurrIDRef`. More details are given in Section 14.7.

14.3.9 Self-documentation

Documentation may be associated to most constructs in a DSD. Documentation is treated as meta-information, which does not affect the processing. It allows a DSD to be virtually self-documenting towards application authors. Also, a DSD processor may use this information when errors are detected to provide the author with useful help.

The DSD language allows three kinds of documentation: `Label`, which can be used to attach a label to the construct; `Doc`, which is intended for full documentation of the construct; and `BriefDoc`, intended for a brief description. Documentation may consist of arbitrary XML, but XHTML is recommended. This allows useful visual effects, such as showing the brief description in a box that pops up when the mouse is over the construct. Examples of documentation are shown in Section 14.6.

14.3.10 The Meta-DSD

The DSD language is self-describable: there is a DSD that completely captures the requirements for an XML document to be a valid DSD. We provide such a DSD of less than 500 lines (allowing sometimes several tags on the same line), called the *meta-DSD*. It can be used both as a human readable description of DSD to clarify its syntax, and by DSD processors to check whether a given XML document is a valid DSD. The meta-DSD resides at <http://www.brics.dk/DSD/dsd.dsd>. Thus, all DSD documents should contain the processing instruction:

```
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>
```

stating that they are intended to conform to the meta-DSD. As noted in Section 14.5, the property of being entirely self-describable is not only aesthetically pleasing, it is also practically useful for application development. Furthermore, it supports development of schemas: the same tool that checks validity of application documents can be used to check that a given XML document is a valid DSD. Most other schema languages require separate tools for that, since they are not completely self-describable.

14.4 The Book Example

We now present a small example of a complete DSD. It describes an XML syntax for databases of books. Such a description could be arbitrarily detailed. We have settled for title, ISBN number, authors (with home pages), publisher (with home page), publication year, and reviews. The main structure of the DSD is as follows:

```
<?dsd URI="http://www.brics.dk/DSD/dsd.dsd"?>

<DSD IDRef="database" DSDVersion="1.0">
  <ElementDef ID="database">
    <ZeroOrMore>
      <Element IDRef="book"/>
    </ZeroOrMore>
    <Element IDRef="database-title"/>
  </ElementDef>
  ...
</DSD>
```

In the database element we use projected content to allow the unique title to appear anywhere in the sequence of book elements. If we wanted to mandate the position of the title element, then a surrounding Sequence constructor was required. The remaining definitions are presented below, excluding the title element and the isbn string type that are shown in Section 14.3. We first show the definition of book elements:

```
<ElementDef ID="book">
  <AttributeDecl Name="isbn" Optional="yes">
    <StringType IDRef="isbn"/>
  </AttributeDecl>
  <Sequence>
    <If><Attribute Name="isbn"/>
      <Then>
        <Optional><Element IDRef="book-title"/></Optional>
      </Then>
      <Else>
        <Element IDRef="book-title"/>
      </Else>
    </If>
    <OneOrMore><Element IDRef="author"/></OneOrMore>
    <Element IDRef="publisher"/>
    <Element Name="year">
      <StringType IDRef="digits"/>
    </Element>
    <Optional>
      <Element Name="review">
        <StringType IDRef="url"/>
      </Element>
    </Optional>
  </Sequence>
</ElementDef>
```

Note that the isbn attribute is optional. If it is not present in a book, then a title is mandatory. The definitions of author and publisher are as follows:

```

<ElementDef ID="author">
  <Sequence>
    <Element Name="first">
      <StringType IDRef="simple"/>
    </Element>
    <Optional>
      <Element Name="initial">
        <StringType IDRef="simple"/>
      </Element>
    </Optional>
    <Element Name="last">
      <StringType IDRef="simple"/>
    </Element>
  </Sequence>
  <Optional><Element IDRef="homepage"/></Optional>
</ElementDef>

<ElementDef ID="publisher">
  <StringType IDRef="simple"/>
  <Optional><Element IDRef="homepage"/></Optional>
</ElementDef>

```

An order is imposed on first, initial, and last, but projected content allows the optional homepage element to appear anywhere. All homepage elements contains a URL:

```

<ElementDef ID="homepage">
  <StringType IDRef="url"/>
</ElementDef>

<StringTypeDef ID="url">
  <ZeroOrMore><AnyChar/></ZeroOrMore>
</StringTypeDef>

```

A naive definition of url is chosen here. It could be replaced with the full 200 line official definition, which is indeed a regular language. The remaining string type definitions are as follows:

```

<StringTypeDef ID="simple">
  <OneOrMore>
    <Union>
      <CharRange Start="a" End="z"/>
      <CharRange Start="A" End="Z"/>
      <CharSet Value=". _ - & " />
    </Union>
  </OneOrMore>
</StringTypeDef>

<StringTypeDef ID="digits">
  <ZeroOrMore>
    <CharRange Start="0" End="9"/>
  </ZeroOrMore>
</StringTypeDef>

```

```

    </ZeroOrMore>
  </StringTypeDef>

```

Such string types could be part of a standard library, constructed as a file containing numerous `StringTypeDef` elements that are accessed through the `include` mechanism. The following definition allows untitled books to receive the default title `Untitled`:

```

<Default>
  <Context>
    <Element Name="book"/>
  </Context>
  <DefaultContent>
    <title>Untitled</title>
  </DefaultContent>
</Default>

```

An example of a conforming application document looks as follows:

```

<?dsd URI="http://www.brics.dk/DSD/book.dsd"?>

<database>
  <title><b>Classic Computer Science Books</b></title>
  <book isbn="0201485419">
    <title>The Art of Computer Programming</title>
    <author>
      <first>Donald</first><initial>E</initial>
      <last>Knuth</last>
      <homepage>
        http://www-cs-faculty.stanford.edu/~knuth/
      </homepage>
    </author>
    <publisher>
      Addison-Wesley
      <homepage>http://www.aw.com</homepage>
    </publisher>
    <year>1998</year>
    <review>
      http://www.amazon.com/exec/obidos/ASIN/0201485419
    </review>
  </book>
</database>

```

14.5 The DSD 1.0 Tool

A prototype DSD processor has been implemented. It tests conformance of application documents and inserts defaults. This shows that it is possible to implement a complete DSD processor in less than 5000 lines of straightforward C code.

Given the URI of an application document containing a DSD-reference processing instruction, the DSD tool performs the traversal of the application document as described in Section 14.3, and if it succeeds, it then performs the ID/IDRef and points-to checks described in Section 14.3.7. Before the application document is processed, the

DSD document, including all application document defaults, is checked to see whether it conforms to the meta-DSD. This check can be omitted by a command-line option if the user is certain that the DSD is in fact valid.

If an error occurs, that is, if a document is not conforming to its DSD, then a suitable error message is inserted in the document which is then output. If the processing succeeds without errors, then the defaults are added to the application document. As an extra feature, the tool can be instructed to add special attributes that detail the element ID assigned to a node. Such parsing information can be useful in subsequent processing by other XML tools.

By using a DSD processor as a front-end for other XML tools, these often become much simpler to construct since the subsequent phases may assume that the input satisfies the syntactic requirements defined by a certain DSD. This is exemplified by the IVR system described in the next section. The DSD processor itself relies on this technique. Using the meta-DSD, which is a complete description of the DSD language itself, the processor checks that a purported DSD document is indeed a DSD. This bootstrapping technique has reduced the size of the implementation considerably and made it more robust and readable.

The DSD processor analyzes application documents in linear time: execution time is proportional to the size of the application document. This assumes that the document size includes the inserted defaults, and that we view all default definitions, including those written in the application document, as belonging to the DSD. The constant of proportionality naturally depends on the complexity of the given DSD. This linear-time property makes the behavior of the DSD processor more predictable than with other schema language processors, where such guarantees are typically not required by the language specification.

14.6 Industrial Case Study: IVR Systems

IVR (Interactive Voice Response) systems range from simple telephony applications to complicated dialogue systems based on speech recognition. But even the simpler systems are notoriously difficult to construct since their programming involves complex timing and error issues. To simplify the task, many layers of abstractions are introduced. At the highest level, an application programmer chooses between ready-made dialogues, which are parameterized with prompts and timeout durations. In this section, we will show how XML and the DSD Schema may help an application programmer to learn and to use a specialized notation with many interdependent parameters such as prompts, timeout values, error counts, and error messages. In particular, we will show how a DSD processor automatically selects defaults for such parameters according to the programmer's preferences.

Our case study is based on XPML (Extensible Phone Markup Language), an HTML-like experimental language developed at AT&T Labs [129]. The XPML notation has evolved from a simple variation of HTML, dubbed PML, to a rather elaborate programming notation for telephone services that rely on text-to-speech, touchtone input, speech recognition, and call control.

Often, XPML documents resemble conventional marked-up documents, but sometimes they are heavily customized with many default time and prompt settings, mak-

ing them more like notations in a programming language. For such markup language applications, DSDs may play an important role in describing almost all syntactic constraints, while providing a practical solution to the handling of defaults. Indeed, questions of how to use PML effectively in practice originally motivated the development of the DSD language.

The XPML notation as outlined here is somewhat incomplete. It is similar to VoiceXML, a new dialogue markup language developed by AT&T, IBM, Lucent and Motorola. VoiceXML is not very similar to HTML, but otherwise resembles XPML in scope and purpose.

14.6.1 The IVR Scenario

Our case study is presented from the application programmer's point of view. The scenario calls for the development of a tiny interactive voice application that greets customers of different nationalities. The programmer will use the domain-specific language XPML, whose syntax is defined using DSD. The main idea of XPML is that simple HTML-like pages describe a finite-state machine, where intra-page hyperlinks become goto statements and text becomes synthesized speech; input fields correspond to subdialogues for obtaining numbers and `select` elements become dialogues à la "for sales, choose 1; for customer service, choose 2,...".

Each subdialogue construct provides numerous parameters for specifying prompts, help messages, timeout durations, timeout counts, and messages in various error situations. As a further complication, there are several interdependencies among these parameters. For example, some HTML elements are associated with several possible *interaction styles* that support situations such as: unusually many choices in a menu, number input restricted to certain ranges, variations in dialogue style ("press any key when you hear the right choice"), etc. The interaction style is specified by an `interaction` attribute. Naturally, the kinds of prompt parameters, along with many other settings, are dependent on the value of this attribute.

14.6.2 DSDs for Syntax Explanations

Our application programmer is a novice XPML user, who has seen only a few examples of XPML code. One role of the DSD is to provide a readable, concise syntactic summary. The programmer should not have to read the DSD as an XML file; instead, a BNF-like version in the form of a hyperlinked HTML document may be produced by an XSLT style sheet transformation. For example, the DSD definition of the element `XPML`, the top element of an XPML document, is shown below (left) through an XSLT style sheet transformation into HTML. The pretty-printed version is designed to resemble the concrete syntax of an application document; the original DSD definition (right) is less appealing:

```
<XPML> ID=XPML:
(  <head>
    Constraint head-constraint
  </head> [Defaultable],
  <body>
    Constraint body-constraint
  </body>)
</XPML>
```

```
<ElementDef ID="XPML">
  <Sequence>
    <BriefDoc>
      The head element
      may be omitted.
    </BriefDoc>
    <Element Name="head"
      Defaultable="yes">
      <Constraint IDRef=
        "head-constraint"/>
    </Element>
    <BriefDoc>
      The body element is
      mandatory.
    </BriefDoc>
    <Element Name="body">
      <Constraint IDRef=
        "body-constraint"/>
    </Element>
  </Sequence>
</ElementDef>
```

The BriefDoc documentation strings of the XML version are translated into HTML title attributes—they provide the effect of a pop-up explanation when the mouse pointer is over the corresponding definition. This particular snippet of a DSD specifies that the XPML element consists of a head element followed by a body element. The head is defaultable, which means that it may be omitted if a default for it has been specified, and its attributes and content are specified by the constraint named head-constraint. Similarly, the body element is specified by the constraint body-constraint. The XSLT style sheet can be found at the DSD Web site; it is rather complicated, taking up approximately 25 pages.

14.6.3 DSDs for Debugging

We now explore how schemas may help the debugging of XML documents. Assume that the application programmer's first attempt at the XPML program is:

```
<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>
    <maintainer address="klarlund@research.att.com"
      loglevel="2"/>
    <title>The Greeting Application</title>
  </head>
  <body>
    Welcome.
    <span nointerrupt="y">
      <audio url="/audioclips/greeting.vox"/>
    </span>
```

```

    <a name="repeat"/>
    <menu name="nationality">
      <option dtmf="0">To end</option>
      <do><a href="#endit"/>
        <comment>go to end point</comment>
      </do>
      <option> If you speak English. </option>
      <do> Hello! How are you? </do>
      <option> If you speak Danish. </option>
      <do> Goddag! Hvordan går det? </do>
    </menu>
    <a href="#repeat"/>
    <a name="endit"/>
  </body>
</XPML>

```

The programmer has inserted a `<?dsd URI="xpml-att.dsd"?>` processing instruction to indicate that the document must conform to the DSD named `xpml-att.dsd`. Using the DSD processor to check the syntax of the document will now produce the response:

```

Error in 'greetings-first-attempt.pml'
line 10: attribute 'nointerrupt' has illegal value 'y'
while checking attribute in constraint
"message-attributes", 'xpml-core.dsd' line 377

```

An automated error analysis tool would display this constraint along with the pertinent auxiliary definitions:

```

ConstraintDef ID=message-attributes:
  nointerrupt="YesOrNo" [Optional]

```

```

StringTypeDef ID=YesOrNo:
  ("yes" | "Yes" | "no" | "No")

```

revealing that the programmer must write "yes" instead of "y". Naturally, the other schema notations offer similar capabilities. Most people will probably get acquainted with schemas only through such error-reporting; thus, it is very important that the schema notation itself is as simple as possible to make error messages understandable for non-experts.

14.6.4 DSDs for Myriads of Defaults

Once the above error is corrected, the DSD processor accepts the document and inserts all the default attributes and default elements specified by the DSD for XPML. The resulting document is:

```

<?dsd URI="xpml-att.dsd"?>
<XPML>
  <head>
    <application name="HELLOWORLD"/>

```

```

    <maintainer address="klarlund@research.att.com"
               loglevel="2"/>
    <title>The Greeting Application</title>
</head>
<body>
    Welcome.
    <span nointerrupt="yes">
        <audio url="/audioclips/greeting.vox"/>
    </span>
    <a name="repeat"/>
    <menu asrmode="none" endchars="#" finaltimeout="5000ms"
          interaction="basic" interdigittimeout="4000ms"
          maxmisselected="3" maxtimeout="2" maxtters="3"
          name="feelings" timeout="0ms">
        <option dtmf="0">To end</option>
        <do><a href="endit"/>
            <comment>go to end point</comment>
        </do>
        <option> If you speak English. </option>
        <do> Hello! How are you? </do>
        <option> If you speak Danish. </option>
        <do> Goddag! Hvordan går det? </do>
        <help>No help is available.</help>
    <initial>
        <enumerate><option/>Press
            <emph><dtmf/></emph>.
        </enumerate>
    </initial>
    <timeout> You have exceeded the time limit. </timeout>
    <toomanyerrors>Sorry, too many errors.</toomanyerrors>
    <counttimeout>Sorry, too many timeouts.</counttimeout>
    <pause>Pausing. Press pound sign to continue.</pause>
</menu>
    <a href="repeat"/>
    <a name="endit"/>
</body>
</XPML>

```

It is similar to the original document except that all timing and counting parameters that are relevant according to the schema have been inserted. Also, various default messages used in error and help situations, like <help>No help is available.</help> have been inserted. Voice programming, as well as HTML layout, is dependent on a great number of parameters whose tuning is often essential to obtaining the desired performance. However, it would be quite a burden if all parameters should explicitly be stated in each document.

This example shows how DSDs generally allow XML notations to be abstracted away from rendering details in a way similar to CSS. However, we should note that DSDs do not subsume CSS: in the domain of visual formatting there are some arithmetic rules about inheritance of values that cannot be expressed in DSD.

DSD style sheets

DSD defaults defined by both the system and the application programmer may be gathered in files known as *external parsed entities*. These are just like XML documents except that multiple root elements are allowed. They work as style sheets by inclusion in the application document via the `include` processing instruction.

Below, the application programmer has defined a DSD style sheet that overrides the default help element for the menu construct in two ways: for a menu without a `class` attribute, the message “We’re sorry, can’t help you more right now, but please call us at 1-800-greetings” is specified; for a menu with a `class` attribute of value `explain`, the default content of help instructs the customer to press “1” to have the error explained.

```
<DSD:Default>
  <Context>
    <Element Name="menu"/>
  </Context>
  <DefaultContent>
    <help>
      We're sorry, can't help you more right now,
      but please call us at 1-800-greetings
    </help>
  </DefaultContent>
</DSD:Default>

<DSD:Default>
  <Context>
    <Element Name="menu">
      <Attribute Name="class" Value="explain"/>
    </Element>
  </Context>
  <DefaultContent>
    <help>
      Press 1 for further explanation.
    </help>
  </DefaultContent>
</DSD:Default>
```

Thus, parameters can be gathered hierarchically in files to achieve the cascading effect that enable abstractions, formulated as sets of defaults, to be easily customized.

14.6.5 DSDs for Simplifying XPML Processing

With a DSD processor, XML documents may be *normalized* by default insertion in the sense that (1) without inserted defaults (assuming all default information is erased from the DSD) the document is not conforming and (2) with defaults inserted the document is conforming and no more defaults would be inserted—if it were to be run again.

Since defaults can only be overridden by application document, the defaults given with the DSD itself provide a set of assumptions about the shape of the document that results from running the DSD processor on a valid document. For example, the XPML interpreter can assume menu elements are fully filled-in with timing attributes

and content such as help and error messages, since an application programmer provided default can change this information, but not permit it to disappear.

For this reason, the system programmer, who is writing a semantic interpreter for XPML, may omit a host of error and default situations that would otherwise be typical of a domain specific language like XPML. In other words, the DSD notation itself, with its emphasis on parameters and defaults, becomes a domain modeling tool that directly simplifies the building of software.

14.6.6 Summary of DSD Advantages

We have made a preliminary description of the full XPML language. Our experiments show that almost all of the syntax and static semantics of XPML can be captured as DSDs. This exercise has illustrated four practical aspects of DSD schemas:

- DSDs aid the XPML programmer to choose the right syntactic constructs. To enhance readability of DSDs, we indicate how to present them in a more conventional BNF-like way that closely resembles the concrete syntax of the XPML notation.
- XPML programmers can easily check their documents for most errors using the DSD processor alone.
- XPML programmers can use the CSS-like default mechanism that comes with DSDs. Thus, XPML programs can be “styled” in a declarative and modular fashion.
- DSD descriptions significantly simplify the programming of an interpreter for XPML.

In contrast, the XML Schema notation proposed by the W3C covers only the first two points, and only partly so: first, the notation is incapable of capturing much of the attribute structure of XPML, and second, the notation itself is so complicated that it may impede its use as an explanatory medium directed towards computer professionals.

14.7 Related Work

The specification of the basic XML notation includes the schema language DTD (Document Type Definition), which is a subset of the DTD mechanism known from SGML. XML DTD is a grammar notation that allows a content model and a list of attributes to be declared for each element name. The content model is a restricted form of regular expressions over element names and chardata nodes: if chardata is allowed, then only unordered content can be described. Also, content specifications have to satisfy a determinism property, which is reminiscent of our operational interpretation of content expressions. Attributes can be declared with another restricted form of regular expressions permitting only enumerations of strings to be specified. The attribute declarations also allow defaults to be specified. As in DSD, validity checking is performed by a top-down traversal of the application document. In addition to grammatical descriptions, DTD also contains the notions of *entity definitions*, which is a kind of macro

mechanism, and *notations*, which provide semantic references to data formats. The typical use of entity definitions is subsumed by our inclusion mechanism and the various definition constructs. DSD, as well as many other schema languages, does not have an equivalent of DTD notations, since we regard them as independent of syntactical descriptions.

It is generally agreed that DTD is insufficient for many purposes. Some typical arguments are: it does not itself use XML notation; most common data types for chardata and attribute values cannot be expressed; there is very limited support for modularity and reuse of descriptions; and content and attribute declarations cannot depend on attributes or element context.

A large number of other schema languages have been proposed since the introduction of XML and DTD. In the following, we give a brief summary of these, focusing on what we believe are the most important alternatives: XML Schema and RELAX NG. A further comparison of various schema language proposals, including DSD, can be found in [146].

14.7.1 XML Schema

Based on the experience with the DTD, XML-Data, DDML, DCD, and SOX schema languages, which we mention in Section 14.7.3, W3C has designed the language XML Schema. The requirements that this schema language should address according to W3C are found in [151]. This document briefly outlines usage scenarios such as publishing, electronic commerce transactions, authoring, databases, and metadata exchange, which are areas we believe are covered by DSDs. The design principles are summarized as follows: “The XML Schema language shall be more expressive than XML DTDs; expressed in XML; self-describing; usable by a wide variety of applications that employ XML; straightforwardly usable on the Internet; optimized for interoperability; simple enough to implement with modest design and run time resources; and coordinated with relevant W3C specs.” Additionally, a number of structural requirements are defined: “The XML Schema language must define mechanisms for constraining document structure (namespaces, elements, attributes) and content (datatypes, entities, notations); mechanisms to enable inheritance for element, attribute, and datatype definitions; mechanism for URI reference to standard semantic understanding of a construct; mechanism for embedded documentation; mechanism for application-specific constraints and descriptions; mechanisms for addressing the evolution of schemata; and mechanisms to enable integration of structural schemas with primitive data types.”

The DSD language, we believe, satisfies the principles and requirements outlined above, except that we have paid less attention to a precise coordination with other W3C standards (some of which were under development when DSD was designed). Laying aside issues such as whether XML Schema or DSD is straightforwardly usable on the Internet, we present next some significant technical and conceptual differences.

Constraints vs. complex types

The most essential difference between XML Schema or DSD is the way structural descriptions are specified. In DSD, the central notion is the *constraint*, which cor-

responds to the *complex types* in XML Schema. However, the constraints in DSD involve boolean logic and context expressions; neither feature has a counterpart in XML Schema.

The DSD constraint mechanism allows attribute declarations and content descriptions to depend on attributes in the current element and of its ancestors. As shown in Section 14.3.2, this is a very useful mechanism that we believe many XML language descriptions can benefit from. In fact, both XHTML and the XML Schema language itself have validity requirements of this form, but they simply cannot be formalized in XML Schema.

Content models

The notion of complex types in XML Schema is also related to our content expressions. In XML Schema, the regular expression operators cannot be combined arbitrarily. For instance, the operator for describing unordered content can contain only individual elements. This makes it difficult to express combinations of ordered and unordered content. Also, in XML Schema when describing mixed content, that is, content containing both elements and chardata, no constraints can be given on the chardata. Only if the content is pure chardata can its values be constrained. In DSD, these limitations do not exist.

String types vs. simple types

The string types in DSD correspond to the *simple types* in XML Schema defined in [25]. While we resort to regular expressions and user defined libraries of common type definitions, XML Schema is primarily based on a large number of predefined types and various derivation mechanisms. But, XML Schema UNIX-style regular expressions are also supported. The derivation operators can admittedly be more appropriate than standard regular expression operators, but the expressive power of these two approaches is formally the same. In our opinion, this sublanguage of XML Schema may be too complex relative to its benefits.

Inclusion and redefinition vs. inheritance and substitution groups

XML Schema and DSD also differ significantly in their approach to amending and reusing definitions. DSD uses a simple inclusion mechanism combined with selective redefinitions, while XML Schema contains a more complicated type system inspired by object-oriented programming languages. This type system contains two mechanisms: inheritance by extension or restriction along with substitution groups. The inheritance mechanism allows instance elements of a subtype in places where a supertype is required, but only if the elements are explicitly typed in the application document using special `xsi:type` attributes. Also, types can be defined to be abstract or final, as known from programming languages. The substitution group mechanism allows groups of types to be defined in a way that resembles the inheritance mechanism, but without the hierarchical type structure and explicit types. The DSD proposal does not rely on object-orientation, since we found that most application domains do not lend themselves to this paradigm and are better served with a simpler mechanism.

This raises the question of how DSDs may emulate inheritance. The answer is that a constraint describing the content of an element type may be extended to include more content according to an attribute describing the subtype. The constraint augmentation technique is much more flexible than derivation, for example, content may depend on more than one attribute. However, it does not offer the guarantee that a later definition will not violate the principle of object-orientation that an object of a subtype can be used wherever a supertype is expected.

Default insertion

The default mechanism in XML Schema is similar to the one of DTDs, except that default strings can be inserted in empty elements. In DSD, the defaults are not tied to the element and attribute declarations, but are instead defined by an association to a context expression that can query ancestors and attributes. If for a default definition this expression is true for the current element, the default is applicable. As shown in Section 14.6.4, this mechanism may be quite useful in practice. Also, with DSDs default content is not inserted when an element is empty, but when a content expression requests it. In contrast to XML Schema, this mechanism allows defaults to be inserted in the middle of a content sequence, and it is not limited to chardata.

Self-describability

In general, a schema language is self-describing if and only if it is possible within that language to express all requirements for a document to define a valid schema. Such a self-description is called a *meta-schema*.

According to the design requirements, XML Schema was originally intended to be self-describable, however, the resulting language is not. As previously mentioned, it is not just a few technicalities that hinder this property: examples as the one in Section 14.3.2 can be found throughout the language specification. This precludes XML Schema from the many practical benefits of having a meta-schema similar to the one for DSD in Section 14.3.10. Additionally, it seems unsatisfactory to suggest an XML language intended to describe all common XML languages that cannot describe itself. In a sense, its complexity is higher than its expressiveness.

It is important to note that any schema language can be tweaked into being self-describable according to the above definition: instead of forbidding certain syntactic constructions, one can allow them all and just give them some obscure but well-defined semantics. The fact that DSD is self-describable does not imply that all valid DSDs are meaningful: since schemas capture requirements only about syntax or static semantics, there may always be semantic inconsistencies in a syntactically valid document. Still, in our experience the meta-DSD is able to catch most errors that occur while writing schemas.

Other features

XML Schema contains a few special features not mentioned so far. The notion of *nil values* in XML Schema allows elements to be empty despite of content requirements. Specifically, such an element must be declared nillable and have a `xsi:nil="true"` attribute. This feature can be emulated directly by our general conditional constraints.

XML Schema includes a subset of XPath [61] for expressing uniqueness requirements, keys, and references. A uniqueness constraint specifies that a given expression must be true at most once throughout a certain subset of the document. Keys and references are similar generalizations of the ID/IDREF concepts used in DTD and DSD. To keep the schema language simple, we have chosen not to include such general mechanisms in DSD.

14.7.2 RELAX NG

As a competitor to W3C's XML Schema, the RELAX NG language has emerged through a joining of the RELAX and TREX projects in an effort sponsored by OASIS. These languages all appeared after the DSD 1.0 specification was published.

RELAX [169] is based on the automata-theoretic characterization of regular tree languages formulated in [167]. According to the original RELAX concept, a specification expresses a nondeterministic bottom-up tree automaton. In order to decide whether a given document is accepted by the automaton, an efficient algorithm must work bottom-up in order to carry out a subset construction on the fly. We depart fundamentally from RELAX on this point: we chose to make DSDs similar to deterministic, top-down automata, even though they formally have less expressive power. There are several reasons for this decision. First, a top-down approach typically matches the hierarchical structure of the information being represented and thus is more natural to use. Second, bottom-up parsing prevents online processing, which requires a left-to-right traversal of the document text. Third, our idea that schemas should be a foundation for extending CSS to arbitrary XML requires that we use the same approach as CSS, which is top-down. Fourth, it is not obvious that the added expressive power is really necessary in practice. With our semantics, defaults are inserted deterministically as a part of the parsing process. Had we chosen a more general automaton model, default insertion would become very complex. Indeed, RELAX is suggested as a notation that is explicitly designed not to support default insertions. RELAX NG has inherited this lack of a default mechanism. But like the XML schema team, we believe that defaults must be supported by the schema notation.

Our notion of constraint assignment is superficially similar to the way automata states are assigned by RELAX to nodes of the XML tree. However, our current semantics is formulated as a parsing process, not in terms of automata theory.

It was announced [168] that the RELAX project, influenced by the DSD notation, would adopt a top-down approach based on an automata-theoretic semantics. This has made the RELAX language quite similar to the TREX language [60], which has motivated the merge of the two projects.

A schema in RELAX NG is described by a top-down grammar, as in DSD. Elements are described by *patterns* corresponding to the notions of constraints in DSD and complex types in XML Schema. In contrast to XML Schema, RELAX NG contains a choice pattern operator, reminiscent of our boolean *Or* operator. However, neither the full boolean logic nor the conditional constraints are available, so complex dependencies may require all combinations of allowed attributes and contents to be spelled out. Also, RELAX NG does not contain a notion equivalent to our context expressions, so ancestor dependencies need to be encoded into the top-down grammar. This can cause a blow-up of the schema description when describing multiple ancestor dependencies.

For instance, to simultaneously disallow nesting anchor elements and form elements, the grammar size will essentially increase by a factor of four.

RELAX NG relies on externally defined data types for attribute values and char-data. Only operators for enumerations and lists are built-in. In current implementations, the data types from XML Schema are supported, but this is not required by the specification. As in XML Schema, chardata cannot be constrained if describing mixed content—in contrast to DSD where this is possible.

We believe that RELAX NG has succeeded in providing a simple and expressive alternative to XML Schema. However, the lack of support for defaults, ancestor dependencies, and boolean logic may limit its usability.

14.7.3 Other Proposals

DDML [32], which also has been called XSchema, was the result of a collaborative effort on the XML-DEV mailing list. It is a relatively straightforward generalization of DTD concepts using an XML notation, only adding little expressive power. A related language called DCD was proposed in [43]. It suggests using COBOL-like pictures for expressing string datatypes, adds cardinality constraints to the content models, and is formulated in the RDF framework. XML-Data [145] introduced element keys and references to generalize the ID/IDREF mechanism from DTD, and also inheritance of element descriptions for supporting modularization and reuse. Related to that is SOX [70], which is based more purely on an object-oriented paradigm. The XDR language [92] was designed as a simplification of XML-Data. None of these languages offer a unifying notion of context-dependent constraint as that in DSD.

Assertion Grammars [184] is an interesting approach that achieves some of our goals since it is based implicitly on nonterminals. It contains a powerful notion of tree patterns, which is reminiscent of our context patterns. Recast in our terminology, assertions are redefinitions of nonterminals that conditionally extend their meaning. The condition reflects the context where the addition is valid. We believe it would be possible to explain Assertion Grammars fully in terms of DSD concepts. Conceivably, Assertion Grammar concepts could be integrated with DSDs, where they would stand for abbreviations of DSD constructs. Assertion Grammars allow only a restricted class of extensions, and they do not allow as flexible context dependencies as DSDs.

The Schematron proposal [119] is based on idea of adapting the XPath framework for expressing tree patterns and validity constraints, as an alternative to grammar-centered formalisms. A Schematron schema consists of a number of declarative rules, each essentially being defined by two XPath expressions: a *context* specifying the applicability of the rule and a *assertion* specifying a validity requirement. Schematron language descriptions are open, in the sense that everything that is not explicitly forbidden is allowed. Most other schema languages, including DSD, have the opposite view. Because of the open description model and the high expressiveness of XPath, Schematron is often viewed as a supplement to ordinary schemas, not as a replacement.

As argued in Section 14.6.4, our form of default insertion is a useful way of assigning CSS-like properties in the form of element attributes to the application document. We know of no other work that has suggested a generalization of CSS based on a schema notation.

Finally, we compare DSDs to XSLT [59], which is a Turing-complete XML transformation language based on a tree-walking model. We have attempted to design DSD such that its expressive power matches essential aspects of this functional programming language. Specifically, it is very common that XSLT programs visit each node only once, recurse on children according to XPath tests that concern attributes of the current node and properties of ancestors, and carry no parameters. Generally speaking, DSDs can mimic the recursion of such XSLT programs. Technically, this can be proven by constructing a DSD constraint for each named or unnamed template. The function `xsl:apply-templates`, the basic recursive construct, can be translated into a constraint that drives typing of subnodes. The nodes selected are identified in the DSD through a propagation of types, where non-empty types are used in all non-selected nodes. The details of this correspondence would require very technical arguments, which are outside the scope of this paper.

In XSLT, the programmer's assumptions about the existence or absence of attributes or children is implicit, and XSLT processors do not produce any error messages if such assumptions are not complied with. With DSD, the assumptions can be formalized and checked using the boolean logic and context-dependent constraint mechanisms.

14.8 Conclusion

The DSD language provides a simple but expressive alternative to other XML schema proposals. It embodies a formal approach to the specification, validation, and default completion of XML syntax. It addresses issues such as context dependencies, declarative defaults, schema evolution, semi-structured data, complex data types, and efficient implementation. It has an expressive power that mirrors some essential aspects of XSLT. Moreover, the DSD language has been implemented and tested in practice. It is our hope that ideas from DSD may further simplify XML standards that go beyond just being grammar notations.

More concretely, we believe that in particular the following ideas have proven successful: the application of context expressions, boolean logic, and conditional constraints to describe dependencies, the flexible content model, the declarative default mechanism, and the top-down traversal method.

By the many proposals for XML schema languages, it is clear that there is no ideal solution to the problem of designing the right schema language that fits all purposes. We believe that the DSD language has succeeded in identifying the most central aspects of defining sets of XML document. Still, from the experience with DSD and other recent schema language proposals we are confident that the DSD language can be further simplified and yet become even more expressive in practice. Based on this, we continue the development of the DSD schema language in the future.² As a first goal, DSD needs proper support for namespaces, as mentioned earlier.

Our implementation of a DSD processor is available in an open source package. Please visit the DSD project home page at <http://www.brics.dk/DSD/> for more information. This home page also contains other DSD resources, such as the official

²The paper [57] contains an informal description of DSD2, a successor to DSD.

specification of the DSD 1.0 language [132], example DSDs and application documents, and the XSLT style sheet for DSDs mentioned in Section 14.6.2.

Acknowledgments

We sincerely appreciate the extraordinarily thorough feedback that we received from the reviewers. We also thank the participants of the Spring 2000 XML course at the University of Aarhus for their enthusiasm. DSD users in the XML community have also provided us with many insightful comments and suggestions.

Chapter 15

Extending Java for High-Level Web Service Construction

with Aske Simon Christensen and Michael I. Schwartzbach

Abstract

We incorporate innovations from the <bigwig> project into the Java language to provide high-level features for Web service programming. The resulting language, **JWIG**, contains an advanced session model and a flexible mechanism for dynamic construction of XML documents, in particular XHTML. To support program development we provide a suite of program analyses that at compile time verify for a given program that no runtime errors can occur while building documents or receiving form input, and that all documents being shown are valid according to the document type definition for XHTML 1.0.

We compare **JWIG** with Servlets and JSP which are widely used Web service development platforms. Our implementation and evaluation of **JWIG** indicate that the language extensions can simplify the program structure and that the analyses are sufficiently fast and precise to be practically useful.

15.1 Introduction

The Java language is a natural choice for developing modern Web services. Its built-in network support, strong security guarantees, concurrency control, and wide-spread deployment in both browsers and servers, together with popular development tools make it relatively easy to create Web services. In particular, JavaServer Pages (JSP) [202] and Servlets [201], which are both Java based technologies, have become immensely popular. However, both JSP, Servlets, and many other similar and widely used technologies, such as ASP, PHP, and CGI/Perl, suffer from some problematic shortcomings, as we will argue in the following and try to address.

15.1.1 Sessions and Web Pages

In general, JSP, Servlets, and related approaches provide only low-level solutions to two central aspects of Web service design: *sessions* and *dynamic construction of Web pages*.

A session is conceptually a sequential thread on the server that has local data, may access data shared with other threads, and can perform several interactions with a client. With standard technologies, sessions must be encoded by hand which is tedious and error prone. More significantly, it is difficult to understand the control-flow of an entire service from the program source since the interactions between the client and the server are distributed among several seemingly unrelated code fragments. This makes maintenance harder for the programmer if the service has a complicated control flow. Also, it prevents compilers from getting a global view of the code to perform whole-service program analyses.

The dynamic construction of Web pages is typically achieved by print statements that piece by piece construct HTML fragments from text strings. Java is a general purpose language with no inherent knowledge of HTML, so there are no compile-time guarantees that the resulting documents are valid HTML. For static pages, it is easy to verify validity [177], but for pages that are dynamically generated with a full programming language, the problem is in general undecidable. Not even the much simpler property of being well-formed can be guaranteed in this way. Instead of using string concatenation, document fragments may be built in a more controlled manner with libraries of tree constructor functions. This automatically ensures well-formedness, but it is more tedious to use and the problem of guaranteeing validity is still not solved.

The inability to automatically extract the control-flow of the sessions in a service raises another problem. Typically, the dynamically generated HTML pages contain input forms allowing the client to submit information back to the server. However, the HTML page with the form is constructed by one piece of the service code, while a different piece takes care of receiving the form input. These two pieces must agree on the input fields that are transmitted, but when the control-flow is unknown to the compiler, this property cannot be statically checked. Thorough and expensive runtime testing is then required, and that still cannot give any guarantees.

The `<bigwig>` language [40] is a research language designed to overcome these problems. Its core is a strongly typed C-like language. On top is a high-level notion of sessions where client interactions resemble remote procedure calls such that the control-flow is explicit in the source code [38]. Also, XHTML [182], the XML version of HTML, is a built-in data type with operations for dynamically constructing documents. The values of this data type are well-formed XHTML fragments which may contain “named gaps”. Such fragments can be combined with a special “plug” operation which inserts one fragment into a gap in another fragment. This proves to be a highly flexible but controlled way of building documents. The client interactions and the dynamic document construction are checked at compile time using a specialized type system [195] and a program analysis [39] performing a conservative approximation of the program behavior to attack the problems mentioned above. More specifically, `<bigwig>` services are verified at compile time to ensure that 1) a plug operation always finds a gap with the specified name in the given fragment, 2) the code that receives form input is presented with the expected fields, and 3) only valid

XHTML 1.0 is ever sent to the clients.

15.1.2 Contributions

In this paper we obtain similar benefits for Java applications. Our specific contributions are to show:

- how the session model and the dynamic document model of <bigwig> can be integrated smoothly into Java;
- how the type system from [195] and the program analysis from [39] can be combined, generalized, and applied to Java to provide even stronger static guarantees than known from <bigwig>; and
- how our service model subsumes and extends both the Servlet and the JSP style of defining Web services.

The integration into Java is achieved using a class library together with some extensions of the language syntax. The resulting language is called **JWIG**. When running a **JWIG** service without applying the static analyses, a number of special runtime errors may occur: If one of the three correctness properties mentioned in the previous section is violated, an exception is thrown. The goal of the static analyses is to ensure at compile time that these exceptions never occur. In addition to having this certainty, we can eliminate the overhead of performing runtime checks.

Such guarantees cannot be given for general Servlet or JSP programs. However, we show that the structures of such programs are special cases in **JWIG**: Both the script-centered and the page-centered styles can be emulated by the session-centered, so none of their benefits are lost.

Our current implementation uses XHTML 1.0, but the approach generalizes in a straightforward manner to an arbitrary interaction language described by an XML schema, such as WML or VoiceXML.

A cornerstone in our program analyses is a novel notion of *summary graphs* which provides a suitable abstraction of the sets of XML fragments that appear at runtime. We show how these graphs can be obtained from a data-flow analysis and that they comprise a precise description of the information needed to verify the correctness properties mentioned above.

Throughout each phase of our program analysis, we will formally define in what sense the phase is correct and we will give a theoretical bound on the worst-case complexity. We expect the reader to be familiar with Java and monotone data-flow analysis, and to have a basic understanding of HTML and XML.

15.1.3 Problems with Existing Approaches

In the following we give a more thorough explanation of the support for sessions and dynamic documents in JSP and Servlets and point out some related problems.

The overall structure of a Web service written as a Servlet resembles that of a CGI script. When a request is received from a client, a thread is started on the server. This thread generates a response, usually an HTML page, and perhaps has some side-effects such as updating a database. Before terminating it sends to the client the response,

```

import javax.servlet.*;
public class SessionServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        ServletContext context = getServletContext();
        HttpSession session = request.getSession(true);
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Servlet</title></head><body>");
        if (session.isNew()) {
            out.println("<form action=\"SessionServlet\">" +
                        "Enter your name: <input name=\"handle\">" +
                        "<input type=\"submit\"></form>");
            session.putValue("state", "1");
        } else {
            String state = (String) session.getValue("state");
            if (state.equals("1")) {
                String name = (String) request.getParameter("handle");
                int users =
                    ((Integer) context.getAttribute("users")).intValue() + 1;
                context.setAttribute("users", new Integer(users));
                session.putValue("name", name);
                out.println("Hello " + name +
                            ", you are user number " + users);
                session.putValue("state", "2");
            } else /* state.equals("2") */ {
                String name = (String) session.getValue("name");
                out.println("Goodbye " + name);
                session.invalidate();
            }
        }
        out.println("</body></html>");
    }
}

```

Figure 15.1: Example Servlet code. The session flow is encoded into the `HttpSession` object, and HTML documents are constructed by printing string fragments to a stream.

which is dynamically created by printing strings to an output stream. We call this a *script-centered* approach. The main advantages of Servlets compared to CGI scripts are higher performance and a convenient API for accessing the HTTP layer. A Servlet engine typically uses a thread pool to avoid the overhead of constantly starting threads. Also, Servlets have the general Java benefits of a sandboxed execution model, support for concurrency control, and access to the large set of available Java packages.

A small Servlet program is shown in Figure 15.1. A client running this service is guided through a sequence of interactions which we call a *session*: First, the service prompts for the client's name, then the name and the total number of invocations is shown, and finally a "goodbye" page is shown. The `ServletContext` object contains information shared to all sessions, while the `HttpSession` object is local to each session. Both kinds of state are accessed via a dictionary interface. The overall structure of the code resembles a `switch` statement that branches according to the current interaction. The Servlet API hides the details of cookies and URL rewriting which is used to track the client throughout the session. HTML documents are generated by printing

```
<html><head><title>JSP Demo</title></head><body>
Hello <%
    String name = request.getParameter("who");
    if (name==null) name = "stranger";
    out.print(name);
%>!
<p>
This page was last updated: <%= new Date() %>
</body></html>
```

Figure 15.2: Example JSP page. Code snippets are embedded within the HTML code using special `<%...%>` tags. When a client requests the page, the code snippets are replaced by the strings that result from the evaluation.

strings to the output stream.

A JSP service turns the picture inside out by being defined by an HTML document with embedded code snippets. We call this a *page-centered* approach. Figure 15.2 shows a simple JSP program which dynamically inserts the current time together with a title and a user name based on the input parameters. This approach is quite similar to ASP and PHP, except that the underlying language and its runtime model is safer and better designed. An implementation of JSP typically performs a simple translation into Servlets. This model fits into situations where the service presents pages that are essentially static but with a few dynamic fragments inserted. For more complicated services the code tends to dominate the pages, such that they converge towards straight Servlet implementations.

Both JSP and Servlets allow only a single interaction with the client before termination. To simulate a sequential thread, the client is given a session identifier that is stored either as a cookie, as an SSL session key, or in a hidden form field. The local state associated with the thread is stored as attributes in an `HttpSession` object from which it must be recovered when execution is later resumed. Thus, a sequence of interactions must be encoded by the programmer in a state machine fashion where transitions correspond to individual interactions. This is somewhat cumbersome and precludes cases where the resident local state includes objects in the heap or a stack of pending method invocations. However, it should be noted that the state machine model implies other advantages when it is applicable. Specifically, it allows robust and efficient implementations as evidenced by J2EE engines that ensure scalability and transaction safety by dividing session interactions into atomic transitions. If a Web service runs on a J2SE engine, then only the disadvantages exist since every interaction is then typically handled by a new thread anyway.

Data that is shared between several session threads must in JSP and Servlets be stored in a dictionary structure or an external databases. This is in many cases adequate, but still conceptually unsatisfying in a language that otherwise supports advanced scoping mechanisms, such as nested classes.

Finally, JSP and Servlets offer little support for the dynamic construction of XHTML or general XML documents. JSP allows the use of static templates in which gaps are filled with strings generated by code fragments. Servlets generate all documents as strings. These techniques cannot capture well-formedness of the generated docu-

ments, let alone validation according to some schema definition. Well-formedness can be ensured by relying on libraries such as JDOM [114], where XML documents are constructed as tree data structures. However, this loses the advantages from JSP of using human readable templates and validity can still only be guaranteed by expensive runtime checks.

Another concern in developing Web services is to provide a separation between the tasks of programmers and designers. With JSP and Servlets, the designer must work through the programmer who maintains exclusive control of the markup tags being generated. This creates a bottleneck in the development process.

The **JWIG** language is designed to attack all of these problems within the Java framework. For a more extensive treatment of Servlets, JSP, and the many related languages, we refer to the overview article on <bigwig> [40]. The central aspects of **JWIG** are an explicit session model and a notion of higher-order XML templates, which we will explain in detail in the following sections.

15.1.4 Outline

We begin by describing the special **JWIG** language constructs and packages in Section 15.2. These extensions are designed to allow flexible, dynamic construction of documents and coherent sessions of client interactions. In Section 15.3 we explain how to obtain **JWIG** program *flow graphs* that abstractly describe the flow of strings and XML templates through programs. These flow graphs form the basis of the data-flow analyses that are described in Section 15.4. They culminate in the inference of *summary graphs* which model the sets of XML documents that variables or expressions may evaluate to at given program points. To compute this information we need a preliminary *string analysis*. In Section 15.5 we describe how the results from the summary graph analysis are used to verify that the runtime errors mentioned earlier cannot occur for a given program. This involves the use of a novel XML schema language, *Document Structure Description 2.0*. We provide an overview of our implementation in Section 15.6, and evaluate it on some benchmark programs to show that the analysis techniques are sufficiently fast and precise to be practically useful. Finally, we describe ideas for future work in Section 15.7.

15.2 The JWIG Language

The **JWIG** language is designed as an extension of Java. This extension consists of a service framework for handling session execution, client interaction and server integration, and some syntactic constructs and support classes for dynamic construction of XML documents.

15.2.1 Program Structure

A **JWIG** application is a subclass of the class `Service` containing a number of fields and inner classes. An instance of this class plays the role of a running *service*.

A service contains a number of different *sessions*, which are defined by inner classes. Each session class contains a `main` method, which is invoked when a client initiates a session. The fields of a service object are then accessible from all sessions.


```

import dk.brics.jwig.runtime.*;

public class MyService extends Service {
    int counter = 0;
    synchronized int next() { return ++counter; }

    public class ExampleSession extends Session {
        XML wrapper =
            [[ <html><head><title>JWIG</title></head>
              <body><[body]></body></html> ]];
        XML hello =
            [[ <form>Enter your name: <input name="handle">
              <input type="submit"></form> ]];
        XML greeting =
            [[ Hello <[who]>, you are user number <[count]> ]];
        XML goodbye =
            [[ Goodbye <[who]> ]];

        public void main() throws IOException {
            show wrapper<[body=hello]>;
            String name = receive handle;
            show wrapper<[body=greeting<[who=name,count=next()]]>;
            exit wrapper<[body=goodbye<[who=name]]>;
        }
    }
}

```

Figure 15.3: Example Jwig program. The MyService service contains one session type named ExampleSession which has the same functionality as the Servlet in Figure 15.1.

This provides a simple shared state that is useful for many applications. Concurrency control is not automatic but can be obtained in the usual manner through synchronized methods. Of course, external databases can also be applied, for instance using JDBC, if larger data sets are in use. The fields of a session object as well as local variables in methods are private to each session thread. This approach of applying the standard scope mechanisms of Java for expressing both shared state and per-session state is obviously simpler than using the ServletContext and HttpSession dictionaries in Servlets and JSP.

Figure 15.3 shows a **Jwig** service which is equivalent to the Servlet service from Figure 15.1. In the following, we describe the new language constructs for defining XML templates, showing documents to the clients, and receiving form input.

Session classes come in a number of different flavors, each with different purposes and capabilities, as indicated by its superclass:

- `Service.Session` is the most general kind of interaction pattern, allowing any number of interactions with the client while retaining an arbitrary session state. When a `Service.Session` subclass is initiated, a new thread is started on the server, which lives throughout all interactions with the client. At all intermediate interactions, after supplying the XML to be sent to the client, the thread simply sleeps, waiting for the client to respond.
- `Service.Page` is used for simple requests from the client. A `Service.Page`

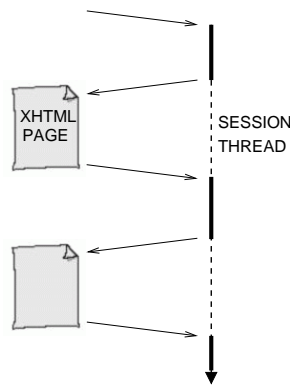


Figure 15.4: Client-server sessions in Web services. On the left is the client's browser, on the right is a session thread running on the server. The thread is initiated by a client request and controls the sequence of session interactions.

is similar to a `Service.Session`, except that it allows no intermediate client interactions. This is conceptually similar to the mechanism used in Servlet and JSP applications, where a short-lived thread is also initiated for each interaction.

- `Service.Seslet` is a special kind of session, called a *seslet*, that is used to interact with the service from applets residing on the client or with other Web services. A `Service.Seslet` does not interact with the client directly in the form of input forms and e.g. XHTML output; instead, it is parameterized with an `InputStream` and an `OutputStream` which are used for communication with the applet or Web service on the client-side. The notion of seslets was introduced in [40].

For the remainder of this article we focus on `Service.Session`. In contrast to the session API in Servlets and JSP, it provides a clear view of the service flow because the sequences of interactions constituting sessions are explicit in the program control-flow. This *session-centered* approach originates from the MAWL project [144].

We specify `Service.Page` by a separate class in order to illustrate that the script- and page-centered approaches are special cases of the session-centered approach, and to identify applications of these simpler interaction models to allow implementations to perform special optimizations.

15.2.2 Client Interaction

Communication with the client is performed through the `show` statement which takes as argument an XML template to be transmitted to the client. A session terminates by executing the `exit` statement whose argument is an XML template that becomes the final page shown. Intermediate XML templates need not conform to any XML schema, but when they are used as arguments to `show` or `exit` statements they must be valid XHTML 1.0 [182]. Otherwise, a `ValidateException` is thrown.

During client interactions, the session thread is suspended on the server. Thus, the execution of the `show` statement behaves as a remote procedure call to the client. Return values are specified by means of form fields in the document. For all form elements,

a default `action` attribute is automatically inserted with a URL pointing to the session thread on the server. The responses from the client are subsequently obtained using the `receive` expression, which takes as argument the name of an input field in the XHTML document that was last shown to the client and returns the corresponding value provided by the client as a `String`. If no such input field was shown to the client, then no corresponding value is transmitted and a `ReceiveException` is thrown. There may be several occurrences of a given input field. In this case, all the corresponding values may be received in order of occurrence in the document into a `String` array using the expression `receive[]`. The non-array version is only permitted if the input field occurs exactly once; otherwise, a `ReceiveException` is thrown. The array version cannot fail: In case there are no fields, the empty array is produced.

Figure 15.4 illustrates the interactions of a session. In the **JWIG** example service in Figure 15.3, the `main` method contains three client interactions: two `show` statements and one `exit` statement. Clearly, the session flow is more explicit than in the corresponding Servlet code in Figure 15.1.

15.2.3 Dynamic Document Construction

In Servlets and JSP, document fragments are generated dynamically by printing strings to a stream. In **JWIG**, we instead use a notion of XML *templates*. A template is a well-formed XML fragment which may contain named *gaps*. A special *plug* operation is used to construct new templates by inserting existing templates or strings into gaps in other templates. These templates are higher-order, because we allow the inserted templates to contain gaps which can be filled in later, in a way that resembles higher-order functions in functional programming languages. Templates are identified by a special data type, `XML`, and may be stored in variables and passed around as any other type. Once a complete XHTML document has been built, it can be used in a `show` statement.

Syntactically, the **JWIG** language introduces the following new expressions for dynamic XML document construction:

<code>[[xml]]</code>	(template constant)
<code>exp₁ <[g = exp₂]</code>	(the plug operator)
<code>([[xml]]) exp</code>	(XML cast)
<code>get url</code>	(runtime template inclusion)

These expressions are used to define template constants, plug templates together, cast values to the `XML` type, and to include template constants at runtime, respectively. The *url* denotes a URL of a template constant located in a separate file, and *xml* is a well-formed XML template according to the following grammar:

<code>xml</code>	<code>: str</code>	(character data)
	<code><name atts> xml </name></code>	(element)
	<code><[g]></code>	(template gap)
	<code><{ stm }></code>	(code gap)
	<code>xml xml</code>	(sequence)
<code>atts</code>	<code>: ε</code>	(empty)

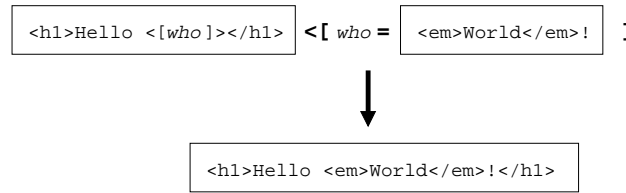


Figure 15.5: The plug operation. The two XHTML templates in the top are combined to produce the one below by plugging into the *who* gap.

<i>name</i> ="str"	(attribute constant)
<i>name</i> =[<i>g</i>]	(attribute gap)
<i>atts</i> <i>atts</i>	(sequence)

Here, *str* denotes an arbitrary Unicode string, *name* an arbitrary identifier, *g* a gap name, and *stm* a statement block that returns a value of type `String` or `XML`. Actual XML values must of course be further constrained to be well-formed according to the XML 1.0 specification [45]. Moreover, in this description we abstract away all DTD information, comments, processing instructions, etc. In Figure 15.3, there are four template constants: *wrapper*, *hello*, *greeting*, and *goodbye*. The *hello* template, for instance, contains two gaps named *who* and *count*, respectively.

XML templates can be composed using the plug operation $exp_1 <[g = exp_2]$. The result of this expression is a copy of exp_1 with all occurrences of the gap named *g* replaced by copies of exp_2 . This is illustrated in Figure 15.5. If exp_2 is a string, all special XML characters (<, >, &, ', and ") are automatically escaped by the corresponding XML character references. If exp_1 contains no gaps named *g*, a `PlugException` is thrown. A gap that has not been plugged is said to be *open*. The exp_2 expression is required to be of type `String` or `XML`. If it is not one of these, it is coerced to `String`. There are three kinds of gaps: template gaps, attribute gaps, and code gaps. Both strings and templates may be plugged into template gaps, but only strings may be plugged into attribute gaps. Attempts to plug templates into attribute gaps will cause a `PlugException` to be thrown. When a template is shown, all remaining open gaps are removed in the following way: Each template gap is replaced by the empty string, and for each attribute gap, the entire attribute containing the gap is removed. As an example, this removal of attributes is particularly useful for checkboxes and radio buttons where checked attributes either must have the value `checked` or be absent. One can use a template containing the attribute gap `checked=[c]` and then plug `checked` into *c* whenever the field should be checked and simply omit the plug otherwise.

The code gaps are not filled in using the plug operation: Instead, when a template containing code gaps is shown, the code blocks in the code gaps are executed in document order. The resulting strings or templates are then inserted in place of the code. Because this does not happen until the template is shown, the code in code gaps can only access variables that are declared in the service class or locally within the code gap.

The plug operation is the only way of manipulating XML templates. Thus, our XML type is quite different from both the string output streams in `Servlets` and `JSP` and the explicit tree structures provided by `JDOM`. We exploit this to obtain a compact and efficient runtime representation, and as a foundation for performing our program

analyses. The notion of code gaps allows us to directly emulate the JSP style of writing services, which is often convenient, but while still having the explicit notion of sessions and the guarantees provided by the program analyses.

In order to be able to perform the static analyses later, we need a simple restriction on the use of forms and input fields in the XML templates: In `input` and `button` elements we require syntactically that attributes named `type` and `multiple` cannot occur as attribute gaps in elements defining input fields. The same restriction holds for `name` attributes, unless the `type` attribute has value `submit` or `image`. In addition, we make a number of simple modifications of the documents being shown and of the field values being received:

1. In HTML and XHTML, lists, tables, and select menus are not allowed to have zero entries. However, it is often inconvenient to be required to perform special actions in those cases. Just before a document is shown, we therefore remove all occurrences of `` and similar constructs. For select menus, we add a dummy option in case none are present.
2. If attempting to receive the selected value of a `select` menu that is not declared as `multiple` or the value of a radio button set, then, if no option is either pre-selected using `checked` or selected by the client, the null value is received instead of throwing a `ReceiveException` — even though no name–value pair is sent according to the XHTML 1.0/HTML 4.01 specification [185].
3. For `submit` and `image` fields, we change the corresponding returned name–value pair from `X=Y` to `submit=X`. This makes it easier to recognize the chosen submit button in case of graphical buttons.
4. For submit buttons, a single name–value pair is produced from the `name` and the `value` attributes. However, for graphical submit buttons, that is, fields with `type="image"`, HTML/XHTML produces two name–value pairs, `X.x` and `X.y` for the click coordinates, where `X` is the value of the `name` attribute, but the `value` attribute is ignored. To obtain a clean semantics, we ensure by patching the returned list of pairs that in every case, all three name–value pairs are produced. For instance, with graphical submit buttons we add a `submit=X` pair, and `submit.x` and `submit.y` contain the click coordinates. For normal submit buttons, `submit.x` and `submit.y` contain the value `-1`.

Clearly, these restrictions and modifications do not impose any practical limitations on the flexibility of the template mechanism. In fact, they serve as a convenience to the programmer since many special cases in XHTML need not be considered.

As for other Java types, casting is required when generic containers or methods are used. An XML template may be cast using the special syntax `([[xml]])exp`. This is a promise from the JWIG programmer that any value that will ever be contained in *exp* at this point will be legal to use in all situations where *xml* is legal. If this is the case, the cast is said to be *valid*. At runtime, a `CastException` is thrown if the sets of gaps and input fields in *exp* do not match those in *xml*. If casting to a template with exactly one occurrence of a given field, then it is required that the actual values also has exactly one occurrence of that field—except for radio buttons, where multiple occurrences count as one. For the gaps, two properties must be satisfied: If the template being cast to has

any gaps of a given name, then at least one such gap must also exist in the actual value; and, if there is a template gap in the template being cast to, then the actual value cannot contain any attribute gaps of that name. Note that this runtime check is not complete since it only considers gaps and input fields and not XHTML validity. However, if invalid XHTML is produced, it will eventually result in a `ValidateException` at a `show` statement.

Alternatively, the ordinary cast `(XML)exp` may be used. It generates a `CastException` if the actual type of `exp` is not XML, but no promises are made about the gaps or input fields.

Large **JWIG** applications may easily involve hundreds of template constants. For this reason, there is support for specifying these externally, instead of inlined in the program source. The construct `get url` loads the XML template located at `url` at runtime. This template can then later be modified and reloaded by the running service.

When the service is analyzed, the template constant referred to by the `get url` construct is loaded and treated as a constant in the analysis. The analysis is then of course only valid as long as the template is unchanged. However, validity will be preserved if the template remains structurally the same. To obtain fresh security guarantees, it is simply required to reinvoke the program analyzer.

These features can also be used to support cooperation between the programmers and the Web page designers. For a first draft, the programmers can create some templates that have the correct structure but only a primitive design. While the program is being developed using these templates, the designers can work on improving them to give them a more sophisticated design. The program analyzer will ensure that the structure of gaps and fields is preserved.

In addition to the main features mentioned above, a session object contains a number of fields and methods that control the current interaction, such as, access control using HTTP authentication, cookies, environment variables, SSL encryption, and various timeout settings. The service object additionally contains a `checkpoint` method which serializes the shared data and stores it on disk. This is complemented by a `rollback` method which can be used in case of server crashes to improve robustness.

To summarize, we have now added special classes and language constructs to support session management, client interactions, and dynamic construction of XHTML documents. By themselves, we believe that these high-level language extensions aid development of Web services. The extensions may cause various exceptions: a `ValidateException` if one attempts to show an XML document which is not valid XHTML 1.0; a `ReceiveException` if trying to receive an input field that occurs an incompatible number of times; a `PlugException` if a plug operation fails because no gaps of the given name and type exist in the template; and a `CastException` if an illegal cast is performed because the gaps or fields do not match. In the following sections, we show that it is possible to statically check whether the first three kinds of exceptions can occur. This is possible only because of the program structure that the new language constructs enforce. For instance, it is far from obvious that similar guarantees could be given for Servlet or JSP services.

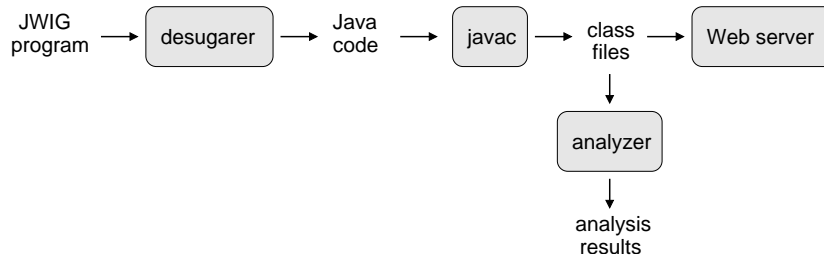


Figure 15.6: The JWIG program translation process in our current implementation. The JWIG program is desugared into Java code which is compiled to class files. These class files are used both in the Web server and to perform the program analyses.

15.2.4 The JWIG Program Translation Process

The steps involved in compiling, analyzing, and running a **JWIG** program are depicted in Figure 15.6. First, the special syntactic constructs of **JWIG** are translated into appropriate Java constructs by a simple source-to-source desugaring transformation. The resulting Java source files are compiled into Java class files as usual. These class files together with the accompanying externally specified XML template constants constitute the Web service. Of course, an implementation is not forced to have this structure: For instance, one could imagine a **JWIG** compiler that directly produces class files instead of going via Java code.

The analysis works on the class file level. When the analyzer is invoked, it is given a collection of class files to analyze. We call this collection the *application classes*; all others constitute the *non-application classes*. Exactly one of the application classes must be a subclass of `Service`. For efficiency reasons, the application classes can be just the few classes that actually constitute the **JWIG** service, not including all the standard Java classes that the program uses. Our analyses are designed to cope with this limited view of the program as an open system.

The soundness of the analyses that we describe in the following sections is based on a set of well-formedness assumptions:

- all invocation sites in the application classes must either always invoke methods in the application classes or always invoke methods in the non-application classes;
- no fields or methods of application classes are accessed by a non-application class;
- no XML operations are performed in non-application classes; and
- XML casts are always valid, according to the definition in the previous section.

These assumptions usually do not limit expressibility in practice. In some cases, the second assumption can be relaxed slightly, for instance if some method called from a non-application class does not modify any `String` or XML value that will ever reach other application class methods. This makes it possible to safely use callback mechanisms such as the `Comparator` interface. The assumption about casts is deliberately

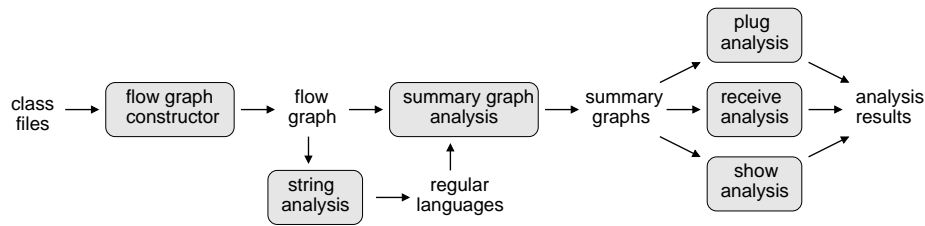


Figure 15.7: Structure of the program analyzer.

quite strong: As ordinary casts, XML casts provide a back-door to the programmer to bypass the static type system.

The structure of the program analyzer is shown in Figure 15.7. From the class files, we first generate *flow graphs*. From these, we generate *summary graphs*, which we analyze in three different ways corresponding to the properties mentioned in the previous section.

15.2.5 An Example Jwig Program

We will use the following **Jwig** program throughout the remaining sections to illustrate the various phases of the analysis. This admittedly rather artificial service applies most **Jwig**-specific language constructs:

```

import dk.brics.jwig.runtime.*;

public class Greetings extends Service {
    String greeting = null;

    public class Welcome extends Session {
        XML cover = [[ <html>
                        <head><title>Welcome</title></head>
                        <body bgcolor=[color]>
                            <{
                                if (greeting==null)
                                    return [[ <em>Hello World!</em> ]];
                                else
                                    return [[ <b><[g]></b> ]] <[g=greeting];
                            }>
                        <[contents]>
                    </body>
                </html> ]];

        XML getinput = [[ <form>Enter today's greeting:
                        <input type="text" name="salutation">
                        <input type="submit"></form> ]];

        XML message = [[ Welcome to <[what]>. ]];

        public void main() {
            XML h = cover<[color="white",contents=message];
            if (greeting==null) {
                show cover<[color="red",contents=getinput];
            }
        }
    }
}

```



```

        greeting = receive salutation;
    }
    exit h<[what=[[<b>BRICS</b>]]]>;
}
}
}

```

The first time the `Welcome` session is initiated, the client is prompted for a greeting text in one interaction, and then in the next interaction the greeting is shown together with a “Welcome to BRICS” message. For subsequent sessions, only the second interaction is performed.

15.3 Flow Graph Construction

Given a **JWIG** program, we first construct an abstract flow graph as a basis for the subsequent data-flow analyses. The flow graph captures the flow of string and XML template values through the program and their uses in `show`, `plug`, and `receive` operations.

15.3.1 Structure of Flow Graphs

The *nodes* in this graph correspond to abstract statements:

<code>x = exp;</code>	(assignment)
<code>show x;</code>	(client interaction)
<code>receive f;</code>	(receive field)
<code>receive[] f;</code>	(receive field array)
<code>nop;</code>	(no operation)

where *exp* denotes an expression of one of the following kinds:

<code>x</code>	(variable read)
<code>"str"</code>	(string constant)
<code>[[xml]]</code>	(XML template constant)
<code>x <[g = y]</code>	(plug operation)
<code>null</code>	(null value)
<code>anystring</code>	(arbitrary string)

and *x* and *y* are program variables, *g* is a gap name, *f* is a field name, *str* is a string constant, and *xml* is an XML template constant that does not contain any code gaps. All code gaps in the original **JWIG** program are expressed using normal gaps and plug operations in the flow graph, as will be explained in Section 15.3.3.

We assume that every variable occurring in the flow graph has a declared type: `STRING` representing strings, or `XML` representing XML templates. These types are extended to expressions as one would expect, and `null` has the special type `NULL`. Let EXP_{STRING} denote the expressions of type `STRING` or `NULL`, and EXP_{XML} denote those of type `XML` or `NULL`.

The assignment statement evaluates its expression and assigns the value to the given variable. The variable and the expression must have the same type. All flow

graph variables are assumed to be declared with a global scope. Evaluating expressions cannot have side-effects. The argument to show statements is always of type XML. As described later, we model receive expressions from the **JWIG** program as pairs of statements, each consisting of a receive statement and an assignment. The `receive` and `receive[]` statements record program locations where input field values are received. The last kind of flow-graph statement, `nop`, is the no-operation statement which we use to model all operations that are irrelevant to our analyses, except for recording split and join points.

The expressions basically correspond to those in concrete **JWIG** programs, except `anystring` which is used to model standard library string operations where we do not know the exact result. In the plug operation, the first variable always has type XML, and the second has type XML or STRING.

Each node in the graph is assigned an *entry label* and an *exit label* as in [174], and additionally each XML template constant has a *template label*. All labels are assumed to be unique. The union of entry labels and exit labels constitute the *program points* of the program.

The graph has two kinds of edges: *flow edges* and *receive edges*. A flow edge models the flow of data values between the program points. Each edge has associated one source node and one destination node, and is labeled with a set of program variables indicating which values that are allowed to flow along that edge. A receive edge goes from a receive node to a show node. Its presence indicates that the control-flow of the program may lead from the corresponding show statement to the receive statement without reaching another show first. We use these edges to describe from which show statements the received field can originate.

15.3.2 Semantics of Flow Graphs

Formally, the semantics of a flow graph is defined by a constraint system. Let V be the set of variables that occur in the flow graph, XML be the set of all XML templates, and $STRING$ be the set of all strings over the Unicode alphabet. Each program point ℓ is associated an environment E_ℓ :

$$E_\ell : V \rightarrow 2^{XML \cup STRING}$$

The entire set of environments forms a lattice ordered by pointwise set inclusion. For each node in the graph we generate a constraint. Let *entry* and *exit* denote the entry and exit labels of a given node. If the statement of the node is an assignment, $x = exp$, then the constraint is:

$$E_{exit}(y) = \begin{cases} \widehat{E}_{entry}(exp) & \text{if } x = y \\ E_{entry}(y) & \text{if } x \neq y \end{cases}$$

For all other nodes, the constraint is:

$$E_{exit} = E_{entry}$$

The map $\hat{E}_\ell : EXP \rightarrow 2^{XML \cup STRING}$ defines the semantics of flow graph expressions given an environment E_ℓ :

$$\hat{E}_\ell(exp) = \begin{cases} E_\ell(x) & \text{if } exp = x \\ \{str\} & \text{if } exp = "str" \\ \{xml\} & \text{if } exp = [[xml]] \\ \pi(E_\ell(x), g, E_\ell(y)) & \text{if } exp = x < [g = y] \\ \emptyset & \text{if } exp = \mathbf{null} \\ STRING & \text{if } exp = \mathbf{anystring} \end{cases}$$

The function π captures the meaning of the plug operation. Due to the previously mentioned type requirements, the first argument to π is always a set of XML template values. The function is defined by:

$$\pi(A, g, B) = \bigcup_{xml \in A, b \in B} \{\bar{\pi}(xml, g, b)\}$$

where $\bar{\pi}$ is defined by induction in the XML template according to the definition in Section 15.2.3:

$$\bar{\pi}(xml, g, b) = \begin{cases} str & \text{if } xml = str \\ <name \ \bar{\pi}(attr, g, b)> & \\ \bar{\pi}(xml', g, b) </name> & \text{if } xml = <name \ attr> xml' </name> \\ b & \text{if } xml = < [g] > \\ < [h] > & \text{if } xml = < [h] > \text{ and } h \neq g \\ \bar{\pi}(xml_1, g, b) \ \bar{\pi}(xml_2, g, b) & \text{if } xml = xml_1 \ xml_2 \end{cases}$$

$$\bar{\pi}(attr, g, b) = \begin{cases} \varepsilon & \text{if } attr = \varepsilon \\ name = "str" & \text{if } attr = name = "str" \\ name = "b" & \text{if } attr = name = [g] \text{ and } b \in STRING \\ name = [h] & \text{if } attr = name = [h] \\ & \text{and } (h \neq g \vee b \in XML) \\ \bar{\pi}(attr_1, g, b) \ \bar{\pi}(attr_2, g, b) & \text{if } attr = attr_1 \ attr_2 \end{cases}$$

This defines plug as a substitution operation where template gaps may contain both strings and templates and string gaps may contain only strings.

For each flow edge from ℓ to ℓ' labeled with a variable x we add the following constraint:

$$E_\ell(x) \subseteq E_{\ell'}(x)$$

to model that the value of x may flow from ℓ to ℓ' .

We now define the semantics of the flow graph as the least solution to the constraint system. This is well-defined because all the constraints are continuous. Note that the environment lattice is not finite, but we do not need to actually compute the solution for any concrete flow graph.

In the following section we specify a translation from **JWIG** programs into flow graphs. In this translation, each `show` statement, `plug` expression, and `receive` expression occurring in the **JWIG** program has a corresponding node in the flow graph. Also, each operand of a `show` or `plug` operation has a corresponding variable in the flow graph. Correctness of such a translation is expressed as two requirements: 1) Let env be the least solution to the flow graph constraint system. If we observe the store of a **JWIG** program at either a `show` or a `plug` operation during some execution, then the value of each operand is contained in $env_\ell(x)$ where ℓ is the node corresponding to the **JWIG** operation and x is the variable corresponding to the operand. 2) If some session thread of an execution of the **JWIG** program passes a `show` statement and later a `receive` expression without passing any other `show` statement in between, then the flow graph contains a receive edge from the node corresponding to the `receive` expression to the node corresponding to the `show` statement.

15.3.3 From JWIG Programs to Flow Graphs

The flow graph must capture the flow of string and XML values in the original **JWIG** program. Compared to `<bigwig>` and the flow graph construction in [39] this is substantially more involved due to the many language features of Java. We divide this data flow into three categories: 1) per-method flow of data in local variables, 2) data flow to and from field variables, and 3) flow of argument and return values for method invocations. Since local variables are guaranteed to be private to each method invocation, we model the first kind of data flow in a control-flow sensitive manner. With field variables, this cannot be done because they may be accessed by other concurrently running session threads, and because we are not able to distinguish between different instantiations of the same class. The second kind of data flow is therefore modeled in a control-flow insensitive manner.

The translation ignores variables whose type is not `String`, XML, or an array of any dimension of these two. For each of the two analyzed types, a unique flow graph *pool variable* is created for representing all the values of that type that cannot be tracked by the analysis. Pooled values include those assigned to and from `Object` variables and arrays, and arguments and results of methods outside the application classes. We add an assignment of `anystring` to the pool variable of type `STRING` to be maximally pessimistic about the string operations in the non-application classes. Something similar is not done for the XML type since we have assumed that XML values are produced only inside the analyzed classes.

In addition to capturing data flow, the flow graph must contain receive edges that reflect the correspondence between `show` and `receive` operations in the **JWIG** program. This requires knowledge of the control flow in addition to the data flow.

Before the actual translation into flow graphs begin, each code gap is converted to a template gap with a unique name, and the code inside the gap is moved to a new method in the service class.

The whole translation of **JWIG** programs into flow graphs proceeds through a sequence of phases, as described in the following. Since **JWIG** includes the entire Java language we attempt to give a brief overview of the translation rather than explain all its details. We claim that this translation is correct according to the definition in the previous section, however it is beyond the scope of this article to state the proof.

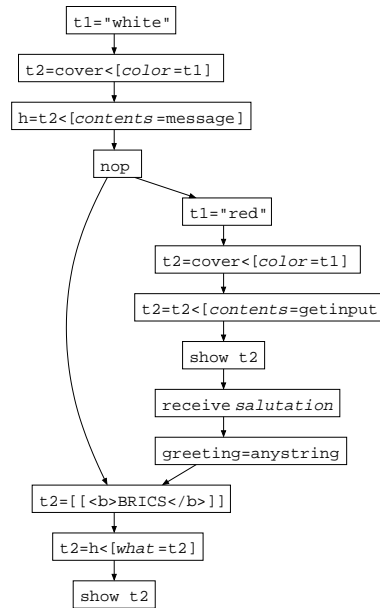


Figure 15.8: Flow graph for the `main` method after Phase 1. All edges are here implicitly labeled with the set of variables $\{h, t1, t2\}$.

1. Individual methods

In the first phase, each method in the application classes is translated individually into a flow graph. Each statement produces one or more nodes, and edges are added to reflect the control flow inside the method. Each edge is labeled with all local variables of the method. Nested expressions are flattened using fresh local variables and assignments. The special **JWIG** operations are translated into the corresponding flow graph statement or expression, and all irrelevant operations are modeled with `nop` nodes. Each `receive` expression is translated into two nodes: a `receive` node and an assignment of `anystring`, since we need to model the locations of these operations but have no knowledge of the values being received. The control structures, `if`, `switch`, `for`, etc., are modeled with `nop` nodes and flow edges, while ignoring the results of the branch conditions. XML casts are translated into XML template constants. This is sound since we have assumed that all casts are valid. Figure 15.8 shows the flow graph for the `main` method of the example **JWIG** program in Section 15.2.5.

2. Code gaps

As mentioned, each code gap has been converted into a template gap whose name uniquely identifies the method containing its code. Before every `show` statement, a sequence of method calls and plug operations is inserted to ensure that all code gaps that occur in the program are executed and that their results are inserted. To handle code gaps that generate templates that themselves contain code gaps, an extra flow edge is added from the end of the plug sequence to the start. The analysis in [39] does not support code gaps.

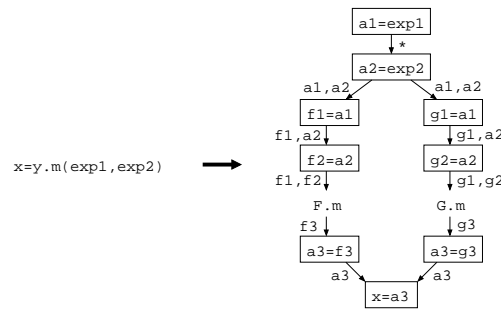


Figure 15.9: Modeling method invocations. Assuming that the invocation of m in the expression on the left may lead to the classes F or G , the flow graph on the right is generated where $F.m$ and $G.m$ are the flow graphs for the target methods. First, the actual parameters are evaluated, then they are passed to the formal parameters for each method and the method bodies are processed, and finally, the return value is collected and the flow is merged. The $*$ label denotes all local variables in the caller method.

3. Method invocations

The methods are combined *monovariently*: each method is represented only once in the flow graph for the whole program. This means that the subsequent analyses that build on the flow graphs also are monovariant. To estimate which methods can be called at each invocation site, a call graph of the **JWIG** program is constructed using a *class hierarchy analysis* (CHA) [71, 203]. This gives us for each invocation site a set of possible target methods. Of course, other call graph analyses could be applied instead, but CHA has proven to be fast and sufficiently precise for these purposes.

For each method invocation, we need to transfer the arguments and the return value, and to add flow edges to and from the possible target methods. The caller method uses a set of special local variables for collecting the arguments and the return value. We first insert a chain of assignments to these caller argument variables. Then we branch for each possible target method and add a chain of assignments from the caller argument variables to the target method parameters, followed by a flow edge to the target method entry point. Similarly, we add flow edges from the method exit points and transfer the return value via a caller result variable. For target methods in non-application classes, we use the pool variables in place of the argument and return value variables of the target method.

Figure 15.9 shows an example of a flow graph for a method invocation where the CHA has determined that there are two possible targets.

4. Exceptions

For every try-catch-finally construct, we add edges from all nodes corresponding to statements in the try block to the entry node of the catch block. These edges are labeled with all local variables of the method. This ensures that the data flow for local variables is modeled correctly. Adding edges from all nodes of the try blocks may seem to cause imprecision. A more complex analysis would only add edges from the nodes that correspond to statements that actually may throw exceptions that are caught

by the `catch` block. However, our simple approach appears to be sufficiently precise in practice.

In order to be able to set up the receive edges in a later phase, we also need to capture the intraprocedural control flow of exceptions. For this purpose, we add a special *drain node* for each method. For each statement, we add a flow edge with an empty label to the drain node of its method. This represents the control flow for uncaught exceptions within each method. This flow may subsequently lead either to drain nodes for caller methods or to `catch` blocks. To model this, we use the CHA information: For each target method of an invocation site, an edge is added from the drain node of the target method to the drain node of the method containing the invocation site. If the invocation site is covered by an exception handler within the method, an extra edge is added from the drain node of the target method to the entry of the handler.

5. Show and receive operations

The preceding phases have set up flow edges representing all possible control flow in the program. This means that we at this point are able to infer the correspondence between `show` and `receive` nodes to create the receive edges by looking at the flow graph alone, without considering the original **JWIG** program.

We treat the entry points of main methods of session classes as if they were `show` statements that show a document with a single form containing no input fields. This models the fact that no input fields can be read with `receive` until a document has been shown.

For each program point ℓ in the flow graph we compute the set of `show` nodes that according to the flow edges may lead to ℓ without passing another `show` node in between. This is done with a fixed point iteration, starting with empty sets at each program point. To each exit label of a `show` node n , we associate the singleton set containing just n . For all other nodes, the sets at the entry points are propagated to the exit points, and at join points, we take the union of the incoming sets. This is essentially a simple forward data-flow analysis on the flow graph. Since there are finitely many `show` nodes and we only apply monotone operations, the iteration eventually terminates. For each `receive` and `receive[]` node, we then add a receive edge to each node in the associated set.

6. Arrays

Array variables are translated into variables of their base type. An array is treated as a single entity whose possible values are the union of its entries. Construction of arrays using `new` is modeled with `null` values to reflect that they are initially empty.

An assignment to an array entry is modeled using weak updating [56] where the existing values of the array are merged with the new value. This is done by inserting two `nop` nodes around the assignment and adding an edge bypassing it labeled by the updated variable. This process is shown in Figure 15.10.

When one array variable is assigned to another, these variables become aliases. Such aliased variables are joined into one variable. This variable will be treated as a

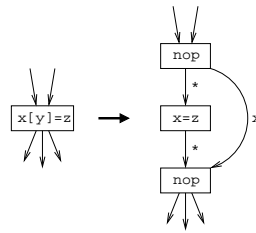


Figure 15.10: Shortcutting array updates. Assignments into arrays are modeled using weak update where all entries are merged.

field variable and handled as described below, if at least one of its original variables was a field variable. This joining is similar to the technique used in [203].

7. Field variables

As mentioned, we model the use of field variables in a flow-insensitive manner where all instances of a given class are merged. This is done for each field simply by adding flow edges labeled by the name of the field from all its definitions to all its uses. To avoid constructing a quadratic number of edges, we add a dummy “ $x=x$ ” node to collect the definitions and the uses for each variable x .

In <bigwig>, a simpler and more restrictive approach was chosen: All global string variables were modeled with `anystring`, and for the global HTML variables, which correspond to the XML field variables in **JWIG**, the initializer expressions would dictate the types [39].

8. Graph simplification

Finally, we perform some reductions of the flow graph. This is not necessary for correctness or precision of the subsequent analyses, but as we show in Section 15.6.2, it substantially decreases the time and space requirements.

First, we remove all code that is unreachable from session entry points according to the flow edges. We ignore edges that originate from method returns since these edges do not indicate reachability.

Using a standard reaching definitions analysis on the flow graph [1, 174], we then find for each assignment all possible uses of that definition. This gives us a set of pairs of nodes where the first node is an assignment to some variable and the second node contains an expression which uses that variable. Once this information is obtained, we remove every flow edge and `nop` node in the graph, and then add new flow edges corresponding to the definition–use pairs. Each new edge is labeled with the single variable of the pair. Finally, a copy propagation optimization is performed to compress chains of copying statements [1].

These transformations all preserve the data flow. In the resulting flow graphs, there are no `nop` nodes and all edges are labeled with a single variable, which is crucial for the performance of the subsequent analyses.

This construction of flow graphs for **JWIG** programs is correct in the sense defined in Section 15.3.2, both with and without the simplification phase.

15.3.4 Complexity

During the construction of the flow graph, we have performed two forward data-flow analyses on the intermediate graphs: one for setting up the receive edges in Phase 5 and the reaching definitions analysis in Phase 8. In the following sections, we will describe two more forward data-flow analyses on flow graphs. To bound the worst-case time requirements for all these analyses, we make some general observations. By implementing the analyses using a standard work-list algorithm rather than the chaotic iteration algorithm [174], the time can be bound by:

$$O\left(t \cdot \sum_{m \in \text{nodes}} |\text{var}(m)| \cdot h \cdot \sum_{m' \in \text{succ}(m)} |\text{var}(m')|\right)$$

where

- t is the maximum cost of computing one binary least-upper-bound operation or one transfer function for a single variable;
- nodes is the set of flow-graph nodes;
- $\text{var}(m)$ denotes the union of the labels of edges adjacent to the node m ;
- h is the height of the lattice for a single variable; and
- $\text{succ}(m)$ is the set of successor nodes of m .

For each node m , an environment associates two lattice elements to each variable v in $\text{var}(m)$, one for the entry label and one for the exit label. Each can change at most h times. Because of the work-list, each change for an exit label can result in at most $\sum_{m' \in \text{succ}(m)} |\text{var}(m')|$ binary least-upper-bound operations and transfer function computations, that is, one for each variable in each successor node, without any other environment changes for exit labels.

Phases 1-7 create at most $O(n^2)$ flow-graph nodes and $O(n^2)$ edges where n is the textual size of the program. The reason for the quadratic increase in the number of nodes is the encoding of argument transferring for method invocations in Phase 3 and the encoding of code gap execution in Phase 2.

The receive edge analysis in Phase 5 does not consider the variables at all. This is equivalent to setting $\text{var}(m)$ to contain just one variable. Since there are at most $O(n^2)$ edges, $\sum_{m \in \text{nodes}} |\text{succ}(m)|$ is $O(n^2)$. The lattice height, h , is the number of `show` statements, which is $O(n)$, and the time t is $O(n)$. Therefore, this analysis runs in time $O(n^4)$.

The complexity of the reaching definitions analysis in Phase 8 can also be bound by the formula above. There are $O(n)$ variables, the lattice height is $O(n)$, and the time t is $O(n)$. Again, since there are $O(n^2)$ edges, $\sum_{m \in \text{nodes}} |\text{succ}(m)|$ is $O(n^2)$. Together, we get that this analysis runs in time $O(n^6)$. Since the other phases of the flow-graph construction are linear in the size of the flow graph, the total construction of the flow graph of a given **JWIG** program requires worst-case $O(n^6)$ time. As shown in Section 15.6.2, this bound is rarely encountered in practice.

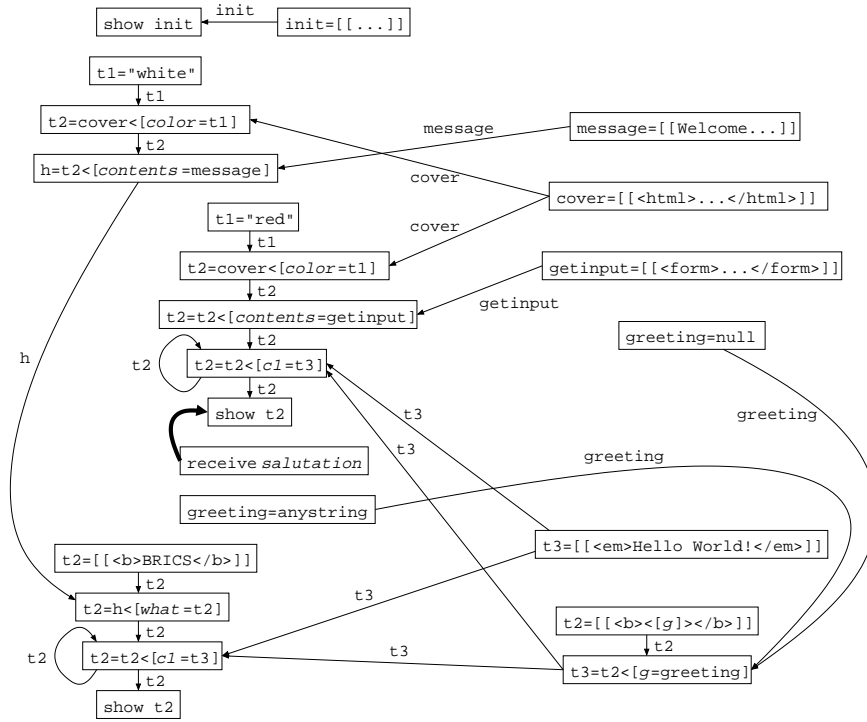


Figure 15.11: Flow graph for the JWIG example program.

After the simplification phase, an extra property is satisfied: Since all flow edges are definition–use edges, $|var(m)|$ is $O(1)$ for all nodes m . Since the flow graph still contains only $O(n^2)$ nodes and edges, the formula above then reduces to:

$$O(n^2 \cdot h \cdot t)$$

This will be used later to estimate the complexities of the remaining analyses.

15.3.5 Flow Graph for the Example

Figure 15.11 shows the flow graph that is generated for the **JWIG** program from Section 15.2.5. The left part of the graph corresponds to the main method, the top right part is the initialization of the field variables, and the bottom right part corresponds to the code in the code gap. The thin edges are flow edges, and the single thick edge is a receive edge. The `show` node in the top left corresponds to the entry point of the session. The `init` template is a simple valid XHTML document with a form that contains no fields. This models the fact that no input values are receivable when session threads are initiated. Note that edges in the part corresponding to the main method have changed compared to Figure 15.8 because of the graph simplification phase.

15.4 Summary Graph Analysis

To statically verify that a given **JWIG** program will never throw any of the special **JWIG** exceptions we perform a *summary graph analysis* based on the flow graph which

contains all the information we need from the original program. Summary graphs model how templates are constructed and used at runtime. This analysis depends on a preliminary *string analysis* that for each string expression finds a regular language that approximates the set of strings it may evaluate to. For each analysis we define a lattice expressing an abstraction of the data values along with a corresponding abstract semantics of the expressions and statements, and then apply standard data-flow analysis techniques to find the least solutions.

15.4.1 String Analysis

Given a flow graph of a **JWIG** program, we must statically model which strings are constructed and used at runtime. In [39] the corresponding analysis is mixed into the summary graph analysis. Separating these analyses leads to a simpler specification and implementation without damaging the analysis precision. We describe here a rather simple analysis which is adequate for all our benchmarks. However, it should be clear that a more precise string analysis capturing relevant string operations easily could be applied instead, as explained in Section 15.7.

We first define a *string environment* lattice:

$$SE = Y \rightarrow REG$$

where Y is the set of string variables that occur in the program and REG is the family of regular languages over the Unicode alphabet. We choose regular languages for modeling string sets because they fit elegantly into the validity analysis algorithm in Section 15.5.5. The ordering on REG is language inclusion and SE inherits this ordering pointwise. We compute an element of this lattice for every program point with a forward data-flow analysis using standard techniques, such as the monotone frameworks of [174, 122]: For every statement that can appear in the flow graph we define a monotone transfer function $SE \rightarrow SE$ and then compute the least fixed point by iteration. First, for each flow-graph string expression we define its abstract denotation by extending every environment map $\Sigma \in SE$ from variables to string expressions, $\widehat{\Sigma} : EXP_{STRING} \rightarrow REG$:

$$\widehat{\Sigma}(exp) = \begin{cases} \Sigma(x) & \text{if } exp = x, \\ \{str\} & \text{if } exp = "str" \\ U^* & \text{if } exp = \mathbf{anyststring} \\ \emptyset & \text{if } exp = \mathbf{null} \end{cases}$$

where U denotes the Unicode alphabet.

For every string assignment statement $x = exp$; the transfer function is defined by:

$$\Sigma \mapsto \Sigma[x \mapsto \widehat{\Sigma}(exp)]$$

that is, the string environment is updated for x to the environment value of exp . Clearly, this is a monotone operation. For all other statements the transfer function is the identity function, since they do not modify any string variables. The lattice is not finite, but by observing that the only languages that occur are either total or subsets of the finitely many string constants that appear in the program, termination of the fixed point

iteration is ensured. A more advanced string analyses, for instance one modeling concatenations more precisely, would require widening for ensuring termination.

The worst-case complexity of this analysis can be estimated by the formula from the previous section. By the observation above, we only use a part of the lattice. This part has height $O(n)$ since there are at most $O(n)$ string constants in the program. The time t for performing a least-upper-bound or transfer function computation is $O(n)$. Thus, this particularly simple string analysis runs in worst-case time $O(n^4)$ where n is the size of the original **JWIG** program.

The result of this analysis is for each program point ℓ a map: $string_\ell : Y \rightarrow REG$. This analysis is correct in the following sense: For any execution of the program, any program point ℓ , and any string variable x , the regular language $string_\ell(x)$ will always contain the value of x at ℓ . That is, the analysis result is a conservative upper approximation of the string flow.

15.4.2 Summary Graphs

As the string analysis, the summary graph analysis fits into standard data-flow frameworks, but it uses a significantly more complex lattice which we define in the following. Let X , G , and N be respectively the sets of template variables, gap names, and template labels that occur in the program. A *summary graph* SG is a finite representation of a set of XML documents defined as:

$$SG = (R, T, S, P)$$

where

- $R \subseteq N$ is a set of *root nodes*,
- $T \subseteq N \times G \times N$ is a set of *template edges*,
- $S : N \times G \rightarrow REG$ is a *string edge* map, and
- $P : G \rightarrow 2^N \times \Gamma \times \Gamma$ is a *gap presence* map.

Here $\Gamma = 2^{\{\text{OPEN}, \text{CLOSED}\}}$ is the *gap presence lattice* whose ordering is set inclusion. Intuitively, the language $L(SG)$ of a summary graph SG is the set of XML documents that can be obtained by unfolding its templates, starting from a root node and plugging templates and strings into gaps according to the edges. Assume that $t : N \rightarrow xml$ maps every template label to the associated template constant. The presence of a template edge $(n_1, g, n_2) \in T$ informally means that the $t(n_2)$ template may be plugged into the g gaps in $t(n_1)$, and a string edge $S(n, g) = L$ means that every string in the regular language L may be plugged into the g gaps in $t(n)$.

The gap presence map, P , specifies for each gap name g which template constants may contain open g gaps reachable from a root and whether g gaps may or must appear somewhere in the unfolding of the graph, either as template gaps or as attribute gaps. The first component of $P(g)$ denotes the set of template constants with open g gaps, and the second and third components describe the presence of template gaps and attribute gaps, respectively. Given such a triple, $P(g)$, we let $nodes(P(g))$ denote the first component. For the other components, the value OPEN means that the gaps may be present, and CLOSED means that they may be absent. Recall from Section 15.2.3 that, at runtime, if a document is shown with open template gaps, these are treated as

empty strings. For open attribute gaps, the entire attribute is removed. We need the gap presence information in the summary graphs to 1) determine where edges should be added when modeling plug operations, 2) model the removal of gaps that remain open when a document is shown, and 3) detect that plug operations may fail because the specified gaps have already been closed.

This unfolding of summary graphs is explained more precisely with the following formalization:

$$\text{unfold}(SG) = \{d \in XML \mid \exists r \in R : SG, r \vdash t(r) \Rightarrow d \text{ where } SG = (R, T, S, P)\}$$

The *unfolding relation*, \Rightarrow , is defined by induction in the structure of the XML template. For the parts that do not involve gaps the definition is a simple recursive traversal:

$$\begin{array}{c} \overline{SG, n \vdash str \Rightarrow str} \\[10pt] \frac{SG, n \vdash xml_1 \Rightarrow xml'_1 \quad SG, n \vdash xml_2 \Rightarrow xml'_2}{SG, n \vdash xml_1 xml_2 \Rightarrow xml'_1 xml'_2} \\[10pt] \frac{SG, n \vdash atts \Rightarrow atts' \quad SG, n \vdash xml \Rightarrow xml'}{SG, n \vdash \langle name \ atts \rangle \ xml \ \langle /name \rangle \Rightarrow \langle name \ atts' \rangle \ xml' \ \langle /name \rangle} \\[10pt] \overline{SG, n \vdash \varepsilon \Rightarrow \varepsilon} \\[10pt] \overline{SG, n \vdash name = "str" \Rightarrow name = "str"} \\[10pt] \frac{SG, n \vdash atts_1 \Rightarrow atts'_1 \quad SG, n \vdash atts_2 \Rightarrow atts'_2}{SG, n \vdash atts_1 \ atts_2 \Rightarrow atts'_1 \ atts'_2} \end{array}$$

There is no unfolding for code gaps since they have already been reduced to template gaps in the flow graph construction. For template gaps we unfold according to the string edges and template edges and check whether the gaps may be open:

$$\begin{array}{c} \frac{str \in S(n, g)}{(R, T, S, P), n \vdash \langle [g] \rangle \Rightarrow str} \\[10pt] \frac{(n, g, m) \in T \quad (R, T, S, P), m \vdash t(m) \Rightarrow xml}{(R, T, S, P), n \vdash \langle [g] \rangle \Rightarrow xml} \\[10pt] \frac{n \in nodes(P(g))}{(R, T, S, P), n \vdash \langle [g] \rangle \Rightarrow \langle [g] \rangle} \end{array}$$

For attribute gaps we unfold according to the string edges, and check whether the gaps may be open:

$$\begin{array}{c} \frac{str \in S(n, g)}{(R, T, S, P), n \vdash name = [g] \Rightarrow name = "str"} \\[10pt] \frac{n \in nodes(P(g))}{(R, T, S, P), n \vdash name = [g] \Rightarrow name = [g]} \end{array}$$

The following function, *close*, is used on the unfolded templates to plug the empty string into remaining template gaps and remove all attributes with gap values:

$$\begin{aligned}
 \text{close}(xml) &= \begin{cases} \langle \text{name } \text{close}(\text{atts}) \rangle & \\ \text{close}(xml') \langle / \text{name} \rangle & \text{if } xml = \langle \text{name } \text{atts} \rangle xml' \langle / \text{name} \rangle \\ \epsilon & \text{if } xml = \langle [g] \rangle \\ \text{close}(xml_1) \text{close}(xml_2) & \text{if } xml = xml_1 xml_2 \\ xml & \text{otherwise} \end{cases} \\
 \text{close}(\text{atts}) &= \begin{cases} \text{close}(\text{atts}_1) \text{close}(\text{atts}_2) & \text{if } \text{atts} = \text{atts}_1 \text{atts}_2 \\ \epsilon & \text{if } \text{atts} = \text{name}=[g] \\ \text{atts} & \text{otherwise} \end{cases}
 \end{aligned}$$

We now define the language of a summary graph by:

$$L(SG) = \{\text{close}(d) \in XML \mid d \in \text{unfold}(SG)\}$$

Let G be the set of all summary graphs. This set is a lattice where the ordering is defined as one would expect:

$$\begin{aligned}
 (R_1, T_1, S_1, P_1) \sqsubseteq (R_2, T_2, S_2, P_2) &\Leftrightarrow \\
 R_1 \subseteq R_2 \wedge T_1 \subseteq T_2 \wedge & \\
 \forall n \in N, g \in G : S_1(n, g) \subseteq S_2(n, g) \wedge P_1(g) \sqsubseteq P_2(g) &
 \end{aligned}$$

where the ordering on gap presence maps is defined by componentwise set inclusion. This ordering respects language inclusion: If $SG_1 \sqsubseteq SG_2$, then $L(SG_1) \subseteq L(SG_2)$.

Compared to the summary graphs in [39] this definition differs in the following ways: First of all, the gap presence map is added. The old algorithm worked under the assumption that all incoming branches to join points in the flow graph would agree on which gaps were open. This was achieved using a simple preliminary program transformation that would convert the “implicit ϵ -plugs” of <bigwig> [35] into explicit ones using the information from the DynDoc type system [195]. Since **JWIG** does not inherit this implicit-plug feature from the <bigwig> design nor uses a DynDoc-like type system we have added the gap presence map. This map contains the information from the “gap track analysis” in [39], but in addition to finding gaps that *may* be open it also tracks *must* information which we need to verify the use of plug operations later.

Secondly, the present definition is more flexible in that it allows strings to be plugged into template gaps. In [39], template gaps were completely separated from attribute gaps. Thirdly, we generalize the flat string lattice to full regular languages allowing us to potentially capture many string operations.

Figure 15.12 shows an example summary graph consisting of two nodes, a single template edge, and two string edges. The language of this summary graph is the set of XML documents that consist of `ul` elements with a `class="large"` attribute and zero or more `li` items containing some text from the language L . Note that the *items* gap in the root template may be OPEN according to the gap presence map, so the empty template may be plugged in here, corresponding to the case where the list is empty.

Note that our analysis is *monovariant*, in the sense that each template constant is only represented once. It is possible to perform a more expensive analysis that

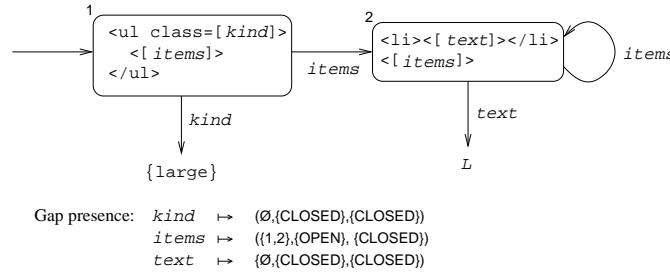


Figure 15.12: A summary graph whose language is a set of XML documents, each containing a `ul` list with zero or more text items and a `class` attribute. The node on the left is a root, and L denotes some set of strings.

duplicates summary graph nodes according to some criteria, but we have not yet encountered the need. On the other hand, our analysis is *polyvariant* in XML element constructors, since these are analyzed separately for each occurrence in the templates.

The summary graph abstraction has evolved through experiments during our previous work on `<bigwig>`. We claim that it is in a finely tuned balance between expressibility and complexity. In [58], we give a constructive proof that summary graphs have the same expressive power as the regular expression types of XDuce [110], in the sense that they characterize the same family of XML languages—if disregarding restrictions on character data and attributes, which are not supported by XDuce. However, summary graphs contain extra structure, for instance, by the gap presence maps, which is required during analysis to model the gap plugging mechanism. Summary graphs contain the structure and expressiveness to capture the intricacies of normal control flow in programs and are yet sufficiently tractable to allow efficient analysis.

15.4.3 Constructing Summary Graphs

At every program point ℓ in the flow graph, each template variable $x \in X$ is associated a summary graph, as modeled by the *summary graph environment* lattice:

$$SGE = X \rightarrow G$$

which inherits its structure pointwise from G . We compute an element of the lattice for every program point using yet another forward data-flow analysis. Let $[[xml]]_n$ mean the template constant labeled n , and let $tgaps(n)$ and $agaps(n)$ be the sets of template gap names and attribute gap names, respectively, that occur in the template constant labeled n . Given an environment lattice element $\Delta \in SGE$ we define an abstract denotation for template expressions, $\hat{\Delta} : EXP_{XML} \rightarrow G$:

$$\hat{\Delta}(exp) = \begin{cases} \Delta(x) & \text{if } exp = x \\ const(tgaps(n), agaps(n), n) & \text{if } exp = [[xml]]_n \\ tplug(\Delta(x), g, \Delta(y)) & \text{if } exp = x <[g = y] \\ & \text{and } y \text{ has type XML} \\ splug(\Delta(x), g, string_\ell(y)) & \text{if } exp = x <[g = y] \\ & \text{and } y \text{ has type STRING} \\ (\emptyset, \emptyset, \lambda(m, h). \emptyset, \lambda h. (\emptyset, \emptyset, \emptyset)) & \text{if } exp = \mathbf{null} \end{cases}$$

where the auxiliary functions are:

$$\begin{aligned}
const(A, B, n) &= (\{n\}, \emptyset, \lambda(m, h). \emptyset, \\
&\quad \lambda h. (\text{if } h \in A \cup B \text{ then } \{n\} \text{ else } \emptyset, \\
&\quad \quad \text{if } h \in A \text{ then } \{\text{OPEN}\} \text{ else } \{\text{CLOSED}\}, \\
&\quad \quad \text{if } h \in B \text{ then } \{\text{OPEN}\} \text{ else } \{\text{CLOSED}\})) \\
\\
tplug((R_1, T_1, S_1, P_1), g, (R_2, T_2, S_2, P_2)) &= \\
& (R_1, \\
& T_1 \cup T_2 \cup \{(n, g, m) \mid n \in nodes(P_1(g)) \wedge m \in R_2\}, \\
& \lambda(m, h). S_1(m, h) \cup S_2(m, h), \\
& \lambda h. \text{if } h = g \text{ then } P_2(h) \text{ else } (p_1 \cup p_2, merge(t_1, t_2), merge(a_1, a_2))) \\
& \quad \text{where } P_1(h) = (p_1, t_1, a_1) \text{ and } P_2(h) = (p_2, t_2, a_2) \\
\\
merge(\gamma_1, \gamma_2) &= \text{if } \gamma_1 = \{\text{OPEN}\} \vee \gamma_2 = \{\text{OPEN}\} \text{ then } \{\text{OPEN}\} \text{ else } \gamma_1 \cup \gamma_2 \\
\\
splug((R, T, S, P), g, L) &= \\
& (R, \\
& T, \\
& \lambda(m, h). \text{if } h = g \wedge m \in nodes(P(h)) \text{ then } S(m, h) \cup L \text{ else } S(m, h), \\
& \lambda h. \text{if } h = g \text{ then } (\emptyset, \{\text{CLOSED}\}, \{\text{CLOSED}\}) \text{ else } P(h))
\end{aligned}$$

For template constants, we look up the set of gaps that appear and construct a simple summary graph with one root and no edges. The *tplug* function models plug operations where the second operand is a template expression. It finds the summary graphs for the two sub-expressions and combines them as follows: The roots are those of the first graph since it represents the outermost template. The template edges become the union of those in the two graphs plus a new edge from each node that may have open gaps of the given name to each root in the second graph. The string edge sets are simply joined without adding new information. For the gaps that are plugged into, we take the gap presence information from the second graph. For the other gaps we use the *merge* function to mark gaps as “definitely open” if they are so in one of the graphs and otherwise take the least upper bound. The *splug* function models plug operations where the second operand is a string expression. It adds the set of strings obtained by the string analysis for the string expression to the appropriate string edge, and then marks the designated gaps as “definitely closed”. The `null` constant is modeled by the empty set of documents. Attempts to plug or show a null template yield null dereference exceptions at runtime, and we do not wish to perform a specific null analysis.

Having defined the analysis of expressions we can now define transfer functions for the statements. As for the other data-flow analysis, only assignments are interesting. For every XML assignment statement $x = exp$; the transfer function is defined by:

$$\Delta \mapsto \Delta[x \mapsto \hat{\Delta}(exp)]$$

and for all other statements the transfer function is the identity function.

By inspecting the *tplug*, *merge*, and *splug* functions it is clear that the transfer function is always monotone. The lattice *SGE* is not finite, but analysis termination

is ensured by the following observation: For any program, all summary graph components are finite, except *REG*. However, the string analysis produces only a finite number of regular languages, and we here use at most all possible unions of these. So, only a finite part of *SGE* is ever used.

The result of this analysis is for each program point ℓ a map:

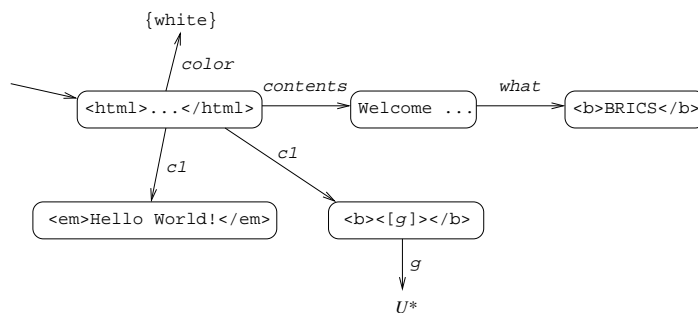
$$\text{summary}_\ell : \text{EXP}_{\text{XML}} \rightarrow G$$

This analysis is conservative as the string analyses, that is, it is sound but not complete: For any execution of the program, any program point ℓ , and any XML expression exp , the set of XML documents $L(\text{summary}_\ell(\text{exp}))$ will always contain the value of exp at ℓ .

The worst-case complexity of this analysis can also be estimated using the formula from Section 15.3.4. The lattice height is the sum of the heights of the four summary graph components. The node set N and the gap name set G both have size $O(n)$, again where n is the size of the original **JWIG** program. The height of the root node component is thus $O(n)$. For each template edge (n, g, n') which is created during the analysis, (n, g) determines a specific gap in a specific template in the original **JWIG** program. Since there can be at most $O(n)$ of these, we can at most construct $O(n^2)$ template edges. Similarly, for the string edge map, all but $O(n)$ pairs of elements from N and G are mapped to a fixed element. For the string analysis, we have argued that the height of the used part of the string lattice is $O(n)$, so the string edge component has height $O(n^2)$. Both the domain and the co-domain of the gap presence map have size $O(n)$, so this component also has height $O(n^2)$. In total, the height h of the summary graph lattice is $O(n^2)$. For the same reasons, the sizes of the summary graphs that are constructed are also at most $O(n^2)$. All operations on summary graphs are linear in their sizes, so the time t for computing a summary graph operation is $O(n^2)$. Inserting this in the formula gives that the summary graph construction runs in time $O(n^6)$ in the size of the program. Note that without the flow-graph simplification phase, the formula would have given $O(n^8)$ instead of $O(n^6)$.

15.4.4 Summary Graphs for the Example

For the **JWIG** example program from Section 15.2.5, the following summary graph is generated for the `exit` statement in the `main` method:



Implicitly in this illustration, the gap presence map maps everything to $(\emptyset, \{\text{CLOSED}\}, \{\text{CLOSED}\})$, and the string edge map maps to the empty language by default. Because

of the simple flow in the example program, the language of this summary graph is precisely the set of XML documents that may occur at runtime. In general, the summary graphs are conservative since they may denote languages that are too large. This means that the subsequent analyses can be sound but not complete.

15.5 Providing Static Guarantees

The remaining analyses are independent of both the original **JWIG** program and the flow graphs. All the relevant information is at this stage contained in the inferred summary graphs. This is a modular approach where the “front-end” and “back-end” analyses may be improved independently of each other. Also, summary graphs provide a good context for giving intuitive error messages.

15.5.1 Plug Analysis

We first validate *plug consistency* of the program, meaning that gaps are always present when subjected to the plug operation and that XML templates are never plugged into attribute gaps. This information is extracted from the summary graph of the template being plugged into.

In the earlier works [195] a similar check was performed directly on the flow graphs. Our new approach has the same precision, even though it relies exclusively on the summary graphs. Furthermore, we no longer require the flow graph to agree on the gap information for all incoming branches in the join points, as mentioned in Section 15.4.2.

For a specific plug operation $x \llbracket g = y \rrbracket$ at a program point ℓ , consider the summary graph $summary_\ell(x) = (R, T, S, P)$ given by the data-flow analysis described in the previous section. Let $(p, t, a) = P(g)$. We now check consistency of the plug operation simply by inspecting that the following condition is satisfied:

$$\begin{aligned} t = \{\text{OPEN}\} \vee a = \{\text{OPEN}\} & \quad \text{if } y \text{ has type STRING, and} \\ t = \{\text{OPEN}\} \wedge a = \{\text{CLOSED}\} & \quad \text{if } y \text{ has type XML.} \end{aligned}$$

This captures the requirement that string plug operations are allowed on all gaps that are present, while template plug operations only are possible for template gaps. If a violation is detected, a precise error message can be generated: for instance, if y has type XML, $t = \{\text{OPEN}\}$, and $a = \{\text{OPEN}, \text{CLOSED}\}$, we report that, although there definitely are open template gaps of the given name, there may also be open attribute gaps, which could result in a `PlugException` at runtime.

As mentioned, the summary graphs that are constructed are conservative with respect to the actual values that appear at runtime. However, the plug analysis clearly introduces no new imprecision, that is, this analysis is both sound and complete with respect to the summary graphs: It determines that a given plug operation cannot fail if and only if for every value in $unfold(summary_\ell(x))$, the plug operation does not fail. If the plug analysis detects no errors, it is guaranteed that no `PlugException` will ever be thrown when running the program. Since the analysis merely inspects the gap presence map component of each summary graph that is associated with a plug operation, this analysis takes time $O(n)$.

15.5.2 Receive Analysis

We now validate *receive consistency*, meaning that `receive` and `receive[]` operations always succeed. For the single-string variant, `receive`, it must be the case that for all program executions, the last document being shown before the receive operation contained exactly one field of the given name. Also, there must have been at least one `show` operation between the initiation of the session thread and the receive operation. If these properties are satisfied, it is guaranteed that no `ReceiveException` will ever be thrown when running the program.

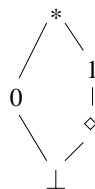
The array variant, `receive[]`, always succeeds, so technically, we do not have to analyze those operations. However, we choose to consider it as an error if we are able to detect that for a given `receive[]` operation, there are no possibility of ever receiving other than the empty array. This is to follow the spirit of Java where, for instance, it is considered a compile-time error to specify a cast operation that is guaranteed to fail for all executions.

In case the name of the field is either `submit`, `submit.x`, or `submit.y`, then we know that it comes from a submit button or image. As described in Section 15.2.3, exactly one value is then always generated. That is, in these cases, both `receive` and `receive[]` always succeed. For the remainder of this section, we thus assume that the field name is not one among those three.

Given a receive operation, we need to count the number of occurrences of input fields of the given name that may appear in every document sent to the client in an associated `show` operation. For a concrete XHTML/HTML document, this information is defined by Section 17.13.2 in [185]. For a running **JWIG** program, a conservative approximation of the information can be extracted from the receive edges in the flow graph and the summary graphs of the associated `show` operations.

Compared to the field analysis in `<bigwig>` [195], this situation differs in a number of ways: 1) The present analysis works on summary graphs rather than on flow graphs. 2) In the old analysis, the plug and receive analyses were combined. We separate them into two independent analyses without losing any precision. 3) In `<bigwig>`, a form is always inserted automatically around the entire document body. That precludes documents from having other forms for submitting input to other services. As described in Section 15.2.2, **JWIG** instead allows multiple forms by identifying those relevant to the session by the absence of an `action` attribute in the form element. 4) The notions of tuples and relations in [195] are in **JWIG** replaced by arrays and `receive[]` operations.

Again, we will define a constraint system for computing the desired information. This information is represented by a value of the following lattice, C :



The element 0 means that there are always zero occurrences of the field, 1 means that there is always exactly one occurrence, * means that the number varies depending on

the unfolding or that it is greater than one, \diamond represents one or more radio buttons, and \perp represents an unknown number. The constraint system applies two special monotone operators on C : \oplus for addition and \otimes for multiplication. These are defined as follows:

\oplus	\perp	0	\diamond	1	*
\perp	\perp	0	\diamond	1	*
0	0	0	\diamond	1	*
\diamond	\diamond	\diamond	\diamond	*	*
1	1	1	*	*	*
*	*	*	*	*	*

\otimes	\perp	0	\diamond	1	*
\perp	\perp	0	\perp	\perp	\perp
0	0	0	0	0	0
\diamond	\perp	0	\diamond	\diamond	*
1	\perp	0	\diamond	1	*
*	\perp	0	*	*	*

Assume that we are given a summary graph (R, T, S, P) corresponding to a specific show statement. Two special functions are used for extracting information about fields and gaps for an individual node in the summary graph:

$$count : N \rightarrow GFP$$

$$allforms : N \rightarrow 2^{GFP}$$

where $GFP = (F \rightarrow C) \times (G \rightarrow C)$ shows the number of occurrences of fields and gaps in a specific form. The *allforms* function returns a set of such values, corresponding to the various forms that may appear, and *count* counts disregarding the form elements:

$$count(n) = (fcount(n, t(n)), gcount(n, t(n)))$$

$$allforms(n) = \bigcup_{k \in forms(t(n))} \{(fcount(n, k), gcount(n, k))\}$$

where

$$forms(xml) = \begin{cases} \{xml\} & \text{if } xml = \langle form \ atts \rangle xml' \ \langle /form \rangle \\ & \text{and } atts \text{ does not contain action} \\ forms(xml') & \text{if } xml = \langle name \ atts \rangle xml' \ \langle /name \rangle \\ & \text{and } name \neq form \\ & \text{or } atts \text{ contains action} \\ forms(xml_1) \cup forms(xml_2) & \text{if } xml = xml_1 \ xml_2 \\ 0 & \text{otherwise} \end{cases}$$

$$gcount(n, xml)(g) = \begin{cases} gcount(n, xml_1)(g) & \text{if } xml = xml_1 \ xml_2 \\ \oplus gcount(n, xml_2)(g) & \text{if } xml = xml_1 \ xml_2 \\ gcount(n, xml')(g) & \text{if } xml = \langle name \ atts \rangle xml' \ \langle /name \rangle \\ \oplus gcount(n, atts)(g) & \text{if } xml = \langle name \ atts \rangle xml' \ \langle /name \rangle \\ 1 & \text{if } xml = \langle [g] \rangle \\ & \text{and } n \notin nodes(P(g)) \\ * & \text{if } xml = \langle [g] \rangle \\ & \text{and } n \in nodes(P(g)) \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned}
gcount(n,atts)(g) &= \begin{cases} gcount(n,atts_1)(g) & \\ \oplus gcount(n,atts_2)(g) & \text{if } atts = atts_1 \ atts_2 \\ 1 & \text{if } atts = \langle [g] \rangle \\ & \text{and } n \notin nodes(P(g)) \\ * & \text{if } atts = \langle [g] \rangle \\ & \text{and } n \in nodes(P(g)) \\ 0 & \text{otherwise} \end{cases} \\
fcount(n,xml)(f) &= \begin{cases} fcount(n,xml_1)(f) & \\ \oplus fcount(n,xml_2)(f) & \text{if } xml = xml_1 \ xml_2 \\ fcount(n,xml')(f) & \text{if } xml = \langle name \ atts \rangle xml' \ \langle /name \rangle \\ & \text{and } name \notin \text{FIELDS} \\ fc(n,atts) & \text{if } xml = \langle name \ atts \rangle xml' \ \langle /name \rangle \\ & \text{and } name \in \text{FIELDS} \\ & \text{and } atts \text{ contains } name="f" \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

The *forms* function finds the relevant form elements in the given template, *gcount* counts the number of occurrences of a given gap name, and *fcount* counts the number of occurrences of a given field name. Note that the latter two functions need to consider the gap presence map of the summary graph. For the field count we can assume that only valid XHTML is shown because of the show analysis presented in the next section, and we can exploit the restrictions about input field elements described in Section 15.2.3. The set $\text{FIELDS} = \{\text{input}, \text{button}, \text{select}, \text{textarea}, \text{object}\}$ contains all names of elements that define input fields. The function $fc(n,atts)$ counts the number of name–value pairs that may be produced: If *atts* contains `type="radio"`, then it returns \diamond ; otherwise, if *atts* contains a type attribute with value `reset`, `submit`, or `image`, or an attribute with name `disabled` or `declare`, it returns 0; otherwise, if it contains `type="checkbox"` or an attribute named `multiple`, it returns $*$, and otherwise it returns 1. In order to detect whether `disabled` or `declare` occur, the gap presence map and the string edges need to be consulted in case of attribute gaps.

With these auxiliary functions in place, we can now define the value $fp \in C$ representing the number of occurrences of *f* in the possible unfoldings of the summary graph:

$$fp = \bigsqcup_{r \in R} \Phi(r)$$

If for every root, the number of occurrences is always 0, always \diamond , or always 1, the final result is 0, \diamond , or 1, respectively; if it sometimes is \diamond and sometimes 1, the result is 1; and otherwise it is $*$. The function Φ traverses the nodes in the summary graph, looking for applicable forms:

$$\Phi(n) = \bigsqcup_{(ff,gg) \in allforms(n)} infields(n,(ff,gg)) \sqcup \bigsqcup_{(n,h,m) \in T, h \in tgaps(n)} \Phi(m)$$

The left-hand factor counts the field occurrences for each form element that occurs directly in the template of *n*, while the right-hand factor looks at the templates that

may be plugged into gaps in n .

$$\text{infields}(n, (ff, gg)) = ff(f) \oplus \bigoplus_{h \in G} (gg(h) \otimes \text{inflow}(n, h))$$

$$\text{inflow}(n, h) = \bigsqcup_{(n, h, m) \in T} \text{infields}(m, \text{count}(m))$$

Given a current node n and an element (ff, gg) of GFP representing the fields and gaps directly available in a particular form, the *infields* function sums the field occurrences according to ff and those that may occur due to plug operations. For the latter part, we iterate through the possible gaps and multiply each count with the gap multiplicity. The *inflow* function follows the template edges and recursively finds the number of field occurrences in the same way as *outflow* but now assuming that we are inside a form.

As usual, we can compute the least fixed point by iteration because the lattice is finite and all operations are monotone. Since the *count* and *allforms* functions never return \perp , the result, fp , is always in the set $\{0, \diamond, 1, *\}$. The desired properties can now be verified by inspecting that:

$$\begin{array}{ll} fp \in \{1, \diamond\} & \text{for receive operations, and} \\ fp \neq 0 & \text{for receive[] operations.} \end{array}$$

for every summary graph computed for some *show* operation that is connected by a receive edge to the receive operation in the flow graph.

As the plug analysis, this receive analysis is both sound and complete with respect to the summary graphs and the receive edges — assuming that only valid XHTML is ever shown: For a *receive* f operation, the analysis determines that it cannot fail if and only if for every label ℓ of a *show* node which has an edge to the *receive* node, it is the case that in every XML document in $L(\text{summary}_\ell(x))$, each form without an *action* attribute produces exactly one f field value. A similar property holds for *receive[]* operations.

For each *receive* and *receive[]* operation, we calculate fp for every summary graph of an associated *show* operation. Thus, fp is calculated $O(n^2)$ times, where n is the size of the original **JWIG** program. The auxiliary functions *count* and *allforms* can be pre-computed in time $O(n)$. Each argument to *infields* denotes a specific form element in a template constant. Since there are $O(n)$ template nodes and $O(n)$ form elements in the program, both Φ and *infields* are given at most $O(n)$ different values as arguments. Since the lattice has constant height, we therefore iterate through the summary graph $O(n)$ times. Each iteration performs a single traversal of the summary graph which takes time $O(n^2)$. In total, the receive analysis runs in time $O(n^5)$ in the size of the original **JWIG** program.

15.5.3 Show Analysis

For every *show* statement in the **JWIG** program, we have computed a summary graph that describes how the XML templates are combined in the program and which XML documents may be shown to the client at that point. This gives us an opportunity to verify that all documents being shown are *valid* with respect to some document type.

In particular, we wish to ensure that the documents are valid XHTML 1.0 [182] which is the most commonly used XML language for interactive Web services. XHTML 1.0 is the official XML version of the popular HTML 4.01. It is relatively easy to translate between the two, so in principle our technique works for HTML as well.

Validity of an XML document means that it is well-formed and in addition satisfies some requirements given by a schema for the particular language. The first part, well-formedness, essentially means that the document directly corresponds to a tree structure whose internal nodes are elements by requiring element tags to balance and nest properly. This part comes for free in **JWIG**, since all XML templates are syntactically required to be well-formed. The remaining validity requirements specify which attributes a given element may have and which text and subelements that may appear immediately below the element in the XML tree. Such properties are specified using a *schema language*. In XHTML, the requirements are given by a DTD (Document Type Definition) schema plus some extra restrictions that cannot be formalized in the DTD language.

Our validation technique is parameterized by the schema description. Thereby we expect that it will be straightforward to support for instance WML or VoiceXML which are used for alternative interaction methods, in place of XHTML. Rather than using DTD, we apply a novel schema language, *Document Structure Description 2.0* (DSD2) [160]. This schema language is more expressive than DTD, so more validity requirements can be formalized. The expressive power of DSD2 is comparable to that of W3C's XML Schema [209], but DSD2 is significantly simpler, as indicated below.

15.5.4 The Document Structure Description 2.0 Language

The DSD2 language is designed as a successor to the schema language described in [132, 133]. A DSD2 schema description of an XML language is itself an XML document. A *DSD2 processor* is a tool that takes as input a DSD2 schema and an XML document called the *instance document*. It then inserts default attributes and contents in the instance document according to the schema and checks whether the instance document with defaults inserted is *valid*. This is done by traversing the tree structure of the instance document in a top-down manner. For every element node, default attributes and contents are inserted and it is checked that all requirements about the attributes and contents of the element are satisfied.

The following description of DSD2 is intended to give a brief overview—not to define the language exhaustively. A normative specification document for DSD2 is currently under development [160].

Conceptually, a DSD2 schema consists of a list of *constraints*. A constraint is either a *declaration*, a *requirement*, a *conditional constraint*, or a *default specification*. Furthermore, there are notions of *string normalization*, *keys* and *references*, and *options* which we can ignore here. During the top-down traversal, the processor checks each element in turn. The *current element* is the one currently being checked. This check of an individual element is performed in five steps:

1. all applicable constraints are found;
2. default attributes and contents are inserted;

3. the requirements are checked;
4. it is checked that all attributes of the current element are declared; and
5. it is checked that the contents matches all contents declarations and that the whole contents are declared.

The following description of the various types of constraints explains these steps in more detail:

Declarations A declaration constraint contains a list of *attribute declarations* and *contents declarations*. An attribute declaration specifies that an attribute with a given name is allowed in the current element provided that the value matches a given regular expression. A contents declaration is a regular expression over characters and element names that specifies a requirement for the presence, ordering, and number of occurrences of sub-elements and character data. A contents declaration only looks at elements that are mentioned in the regular expression. This subsequence of the contents is called the *projected contents*. If the expression contains any character sub-expressions, all character data in the contents is included in the projected contents. Checking a contents declaration succeeds if the projected contents matches the regular expression. All attributes and contents that have been matched by a declaration are considered to be *declared*.

Requirements A requirement constraint contains boolean expressions that must evaluate to true for the current element. Boolean expressions are built of the usual boolean operators, together with attribute expressions which probe the presence and values of attributes, element expressions which probe the name of the current element, and *parent* and *ancestor* operators which probe whether certain properties are satisfied for the elements above the current element in the instance document tree.

Conditional constraints A conditional constraint contains a list of constraints whose applicability is guarded by a boolean expression. Only when the boolean expression evaluates to true for the current element, the constraints within are processed.

Defaults A default constraint specifies a default value for an attribute or a default contents sequence for an empty element. In case of conflicts, for instance if two default attributes with the same name are applicable, all but the one specified last in the schema are ignored.

For convenience, specifications can be grouped and named for modularity and reuse. Furthermore, the DSD2 schema can restrict the name of the root element: for example, in XHTML, it must be `html`. DSD2 has full support for Namespaces [44]. For XHTML, the namespace `http://www.w3.org/1999/xhtml` is used, and all elements must use the empty namespace prefix.

As an example, the following snippet of the DSD2 description of XHTML 1.0 describes `d1` elements:

```

<if><element name="dl"/>
  <declare>
    <attribute name="compact"><string value="compact"/></attribute>
    <repeat min="1"><union>
      <element name="dt"/><element name="dd"/>
    </union></repeat>
  </declare>
  <constraint ref="ATTRS"/>
</if>

```

These constraints show that a `dl` element may contain a `compact` attribute, provided that its value is `compact`, and that it must contain at least one `dt` or `dd` sub-element. Additionally, `ATTRS`, which is defined elsewhere, describes some additional common attributes that may occur.

The following example (abbreviated with “...”) describes a elements:

```

<if><element name="a"/>
  <declare>
    <attribute name="name"><stringtype ref="NMTOKEN"/></attribute>
    <attribute name="shape"><stringtype ref="SHAPE"/></attribute>
    ...
    <repeat><union>
      <string/>
      <contenttype ref="PHRASE"/>
      ...
    </union></repeat>
  </declare>
  <constraint ref="HREFLANG"/>
  ...
  <default name="shape" value="rect"/>
  <require>
    <not><ancestor><element name="a"/></ancestor></not>
  </require>
</if>

```

This reads: If the current element is named “a”, then the sub-constraints are applicable. First, the attributes `name`, `shape`, etc. are declared. The `stringtype` constructs are references to regular expressions defining the valid attribute values. Then, a contents declaration states that all text is allowed as contents together with some contents expressions defined elsewhere. After that, there are some references to constraint definitions and a default specification for the `shape` attribute. Finally, there is a requirement stating that a elements cannot be nested. The latter constraint is an example of a validity requirement that cannot be expressed by DTD or XML Schema.

As a final example, the following requirement can be found in the description of input elements:

```

<require>
  <or>
    <attribute name="type">
      <union><string value="submit"/><string value="reset"/></union>
    </attribute>
    <attribute name="name"/>
  </or>
</require>

```

This states that there must be a type attribute with value `submit` or `reset` or a name attribute. This is another validity requirement that cannot be expressed concisely in most other schema languages. The whole DSD2 schema for XHTML 1.0 can be found at <http://www.brics.dk/DSD/xhtml1-transitional.dsd>.

15.5.5 Validity Analysis

We show below how the DSD2 processing algorithm explained in the previous section generalizes from concrete XML documents to summary graphs. In fact, the DSD2 language has been designed with summary graph validation in mind. Since the DSD2 language is a generalization of the DTD language, the following algorithm could be adapted to DTD. One benefit of using DSD2 instead of DTD or XML Schema is that it allows us to capture many more errors. Every validity requirement that merely appear as comments in the DTD schema for XHTML can be formalized in DSD2, as exemplified in the previous section. Of course, there still are syntactic requirements that even DSD2 cannot express. For instance, in tables, the `thead`, `tfoot`, and `tbody` sections must contain the same number of columns, and the `name` attributes of a elements must have unique values. Summary graphs clearly do not have the power to count such numbers of occurrences, so we do not attempt to also check these requirements. Still, our approach based on DSD2 captures more syntactic errors than is possible with DTD. Only the uniqueness requirements specified by `ID` and `IDREF` attributes are not checked, but they do not play a central role in the schema for XHTML anyway.

Recall from Section 15.2.3 that we make a few modification of the documents at runtime just before they are sent to the clients. It is trivial to modify the validation algorithm to take these modifications into account. For instance, rather than requiring one or more entries in all lists, the analysis permits any number since lists with zero entries are always removed anyway.

Given a DSD2 schema and a summary graph $SG = (R, T, S, P)$ associated to some show statement, we must check that every XML document in $L(SG)$ is valid according to the schema. The algorithm for validating a summary graph with respect to a DSD2 schema proceeds in a top-down manner mimicking the definition of the unfolding relation in Section 15.4.2, starting from the root elements in the templates of the summary graph root nodes. In contrast to the analyses described in the previous sections, we will describe this one in a less formal manner because of the many technical details involved. Rather than showing all the complex underlying equation systems or describing the entire algorithm in detailed pseudo-code, we attempt to present an concise overview of its structure.

For each root $r \in R$, we perform the following steps:

1. Let p be a pointer to the root element in $t(r)$.
2. Check that the name of the p element is valid for a root element.
3. Initialize $M = \emptyset$ and call $check(r, p, c_0)$.

If no violations are detected, SG is valid which implies that all documents in $L(SG)$ are valid. The *check* procedure is given a summary graph node n , a pointer p to an element in $t(n)$, and a *context map* c . It recursively checks validity of the p element

and the sub-trees below it, assuming the specified context. A context map assigns a truth value to every parent and ancestor expression occurring in the schema. The map c_0 maps everything to *false*.

The set M is a global *memoization set* which consists of triples that—as the arguments to *check*—contain a summary graph node, a pointer to an element in the template of the node, and a context map. This set at all times describes the elements and contexts that already have been checked. We use it to avoid performing redundant computations. For a given schema and summary graph, there are only finitely many possible memoization sets, so termination is guaranteed, even in case of cycles in the summary graph.

The procedure $check(n, p, c)$ does the following, essentially corresponding to the five steps for checking individual elements in the algorithm from the previous section, but now also considering gaps:

1. If $(n, p, c) \in M$, then skip all the following steps; otherwise add (n, p, c) to M and proceed.
2. Compute a new context map c' . This is done by re-evaluating each parent and ancestor expression, based on the p element and the c map. Note that because of the c map, this can be done in constant time in the size of the summary graph and the templates.
3. Find all constraints in the schema that are applicable for the p element. This is done by traversing the schema and for each conditional constraint, evaluating its boolean expression. If it evaluates to *true*, it is included. When evaluating boolean expressions, the context map c' is consulted for evaluating parent and ancestor sub-expressions. We describe later more details on how the boolean expressions are evaluated.
4. Build a collection of defaults that should be inserted. This is done by traversing the applicable defaults. An attribute default is included if the p element does not have an attribute of that name. A contents default is included if the contents of the p element is empty. In order to determine whether or not attributes or contents occur, we need to consider the gap presence map, P , the template edges, T , and the string edges, S , in case there are gaps among the attributes or contents of the p element. Note that we do not actually insert the defaults here but only collect them. When the defaults are collected, we need to update c' according to the collected defaults since default attributes may affect the truth values of the parent and ancestor expressions.
5. For each applicable requirement, check that its boolean expression evaluates to *true*. If it does not, then the summary graph is not valid.
6. For each attribute in the p element, including the collected default attributes, check that it is declared by the applicable attribute declarations. An attribute is declared by an attribute declaration if the attribute value matches the regular expression of the declaration. In case the attribute is an attribute gap, this amounts to checking inclusion of one regular language of Unicode strings in another. However, it is possible that one attribute declaration specifies that a given

attribute may have one set of values and another declaration specifies that the same attribute may also have another set of values. Therefore, in general, we check that all values that are possible according to the string edges, S , match some declaration. If some attribute is not declared, the summary graph is not valid.

7. For each contents declaration, check that the contents of p element matches the declaration. As previously mentioned, a contents sequence matches a contents declaration if the projected contents is in the language of the regular expression of the declaration. In case there are no gaps in the contents, this is a simple check of string inclusion in a regular language. If there are gaps, the situation is more involved: The template edges from the gaps may lead to templates which at the top-level themselves contain gaps. (The *top-level* of a template is the sequence of elements, characters, and gaps that are not enclosed by other elements.) In turn, this may cause loops of template edges. Therefore, in general, the set of possible contents sequences forms a context-free language, which we represent by a context-free grammar. Without such loops, the language is regular. The problem of deciding inclusion of a context-free language in a regular language is decidable [108], but computationally expensive. For that reason, we approximate the context-free language by a regular one: Whenever a loop is found in the context-free grammar, we replace it by the regular language A^* where A consists of the individual characters and elements occurring in the loop. This allows us to apply a simpler regular language inclusion algorithm. Although loops in summary graph often occur, our experiments show that it is rare that this approximation actually causes any imprecision. In addition to checking that all contents declarations are satisfied, we check that all parts of the contents has been declared, that is, matched by some declaration. If not, the summary graph is not valid. Again, if any declaration contains a character sub-expression, all character data is considered declared.
8. Look through the immediate contents of the p element. For each sub-element p' that occurs within $t(n)$, call $check(n, p', c')$ recursively. For each template gap g in the contents, find all each outgoing template edge $(n, g, n') \in T$ and call $check(n', p', c')$ recursively for every element p' occurring at the top-level of $t(n')$.

Note that, assuming a fixed schema, steps (1)-(7) can be performed in linear time in the number of attributes in the current element and in the length of its contents, independently of the rest of the $t(n)$ template and of all the other templates. Because of the memoization, the entire algorithm for whole summary graphs therefore runs in linear time in the size of the templates, which in a sense is optimal.

Also note that in contrast to the algorithm for validating XML documents, this one does not have side-effects on the summary graphs: defaults are *not* inserted in the templates, but the validity checks works as if they were inserted. It would not be possible to explicitly insert the defaults, since each template in general is evaluated in many different contexts during the validity check.

Evaluation of boolean expressions for conditional constraints and requirement constraints is non-trivial because we have to consider all the possible unfoldings of the

summary graph. We apply a *four-valued* logic for that purpose. Evaluating a boolean expression results in one of the following values:

- true* – if evaluating the expression on every possible unfolding would result in *true*;
- false* – if they all would result in *false*;
- some* – if some unfolding would result in *true* and others in *false*;
- don't-know* – if the processor is unable to detect whether all, no, or some unfoldings would result in *true*.

All boolean operators extend naturally to these values. The value *don't-know* is for instance produced by the conjunction of *some* and *some*. If the guard of a conditional constraint evaluates to *don't-know*, we terminate with a “don't know” message. However, for our concrete XHTML schema, this can in fact never happen.

Compared with the technique described in [39], we have now moved from an abstract version of DTD to the more powerful DSD2 schema language. Furthermore, by the introduction of four-valued logic for evaluation of boolean expressions, we have repaired a defect that in rare cases caused the old algorithm to fail.

Our algorithm is sound, that is, if it validates a given summary graph it is certain that all unfoldings into concrete XML documents are also valid. Because of the possibility of returning “don't know” and of the approximation of context-free languages by regular ones, the algorithm is generally not complete. An alternative to our approach of “giving up” when these situations occur would be to branch out on all individual unfoldings and use classical two-valued logic. This shows that the problem is decidable. However, the complexity of the algorithm would then increase significantly, and, for the XHTML schema, false errors can only occur in the rare cases where we actually need to approximate context-free languages by regular ones, as mentioned above.

15.6 Implementation and Evaluation

To make experiments for evaluating the **JWIG** language design and the performance of the program analyses, we have made a prototype implementation. It consists of the following components, roughly corresponding to the structure in Figures 15.6 and 15.7:

- a simple desugarer, made with JFlex [141], for translating **JWIG** programs into Java code;
- the <bigwig> low-level runtime system [38] which in its newest version [170] is based on a module for the Apache Web Server [20] and extended with Java support;
- a Java-based runtime system for representing and manipulating XML templates;
- a part of the Soot optimization framework for Java [213, 203], which converts Java bytecode to a more convenient 3-address instruction code language, called Jimple;
- a flow graph package, which operates on the Jimple code generated by Soot and also uses Soot's CHA implementation;

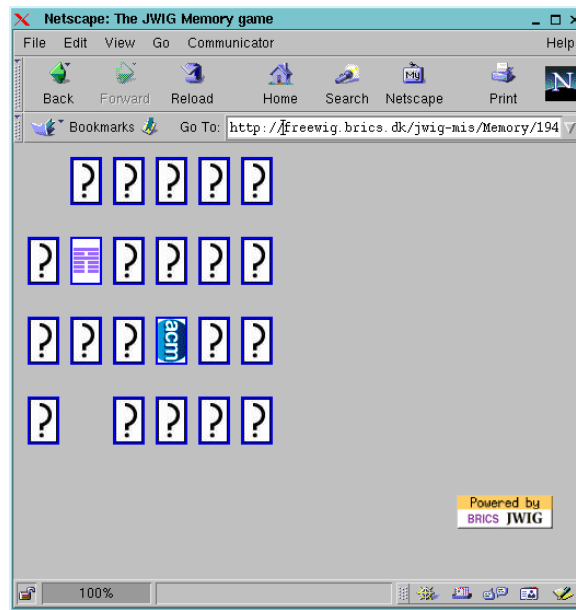


Figure 15.13: A snapshot of the Memory Game being played.

- a finite-state automaton package with UTF16 Unicode alphabet and support for all standard regular operations [171];
- a summary graph construction package which also performs the string analysis;
- a plug and receive analyzer which performs the checks described in Section 15.5.1 and 15.5.2; and
- a DSD2 validity checker with summary graph support.

All parts are written in Java, except the low-level runtime system which is written in C. The Java part of the runtime system amounts to 3000 lines, and the analysis framework is 12,500 lines.

15.6.1 Example: The Memory Game

To give a more complete example of a **JWIG** service, we present the well-known Memory Game, where the player must match up pairs of cards lying face down. First, the number of pairs is chosen, next the game itself proceeds iteratively, and finally the player is congratulated. A snapshot of the game in progress is seen in Figure 15.13.

The main session, presented in Figure 15.14, looks just like a corresponding sequential Java program. The templates being used are presented in Figure 15.15. The construction of a grid of cards is performed by the `makeCardTable` method presented in Figure 15.16. The class representing individual cards is seen in Figure 15.17. In all, the Memory Game is written in 169 lines of **JWIG**.

By itself, the session concept and the XML templates simplify the program compared to solutions in JSP or Servlets. Furthermore, since the example is analyzed

```

public class Game extends Session {
    public void main() {
        // ask for number of cards
        int howmany;
        do {
            show wrap <[ body = welcome <[ atmost = images.length ]]];
            howmany = Integer.parseInt(receive howmany);
        } while (howmany < 1 || howmany > images.length);

        // generate random permutation of cards
        Card[] cards = new Card[howmany*2];
        Random random = new Random();
        for (int i = 0 ; i < howmany ; i++) {
            for (int c = 0 ; c < 2 ; c++) {
                int index;
                do {
                    index = random.nextInt(howmany*2);
                } while (cards[index] != null);
                cards[index] = new Card(i);
            }
        }

        // play the game
        int pairsleft = howmany;
        int moves = 0;
        show makeCardTable(cards);
        while (pairsleft > 0) {
            // first card picked
            int firstcard = Integer.parseInt(receive submit);
            cards[firstcard].status = 1;
            show makeCardTable(cards);

            // second card picked
            int secondcard = Integer.parseInt(receive submit);
            cards[secondcard].status = 1;
            moves++;

            // check match
            if (cards[firstcard].value == cards[secondcard].value) {
                cards[firstcard].status = 2;
                cards[secondcard].status = 2;
                if (--pairsleft > 0)
                    show makeCardTable(cards);
            } else {
                show makeCardTable(cards);
                cards[firstcard].status = 0;
                cards[secondcard].status = 0;
            }
        }

        // done, show result
        exit farewell <[ howmany = howmany, moves = moves ];
    }
}

```

Figure 15.14: The main session of the Memory Game.

```

private static final XML wrap = [[
    <html>
        <head><title>The JWIG Memory game</title></head>
        <body><form><[body]></form></body>
    </html>
]];

private static final XML welcome = [[
    <h3>Welcome to the JWIG Memory game!</h3>
    <p>How many pairs of cards do you want (from 1 to <[atmost]>)?</p>
    <input type="text" name="howmany"/>
]];

private static final XML farewell = wrap <[ body = [[
    <h3>Thank you for playing this game!</h3>
    <p>You found all <[howmany]> pairs using <[moves]> moves.</p>
]] ];

```

Figure 15.15: Templates from the Memory Game.

```

private XML makeCardTable(Card[] cards) {
    XML table = [[ <table><[row]></table> ]];
    for (int y=0; y < (cards.length+COLS-1)/COLS; y++) {
        XML row = [[ <tr><[elem]></tr><[row]> ]];
        for (int x=0; x < COLS; x++) {
            XML elem = [[ <td><[contents]></td><[elem]> ]];
            int index = y*COLS+x;
            if (index < cards.length) {
                elem = elem <[ contents = cards[index].makeCard(index) ];
            }
            row = row <[ elem = elem ];
        }
        table = table <[ row = row ];
    }
    return wrap <[ body = table ];
}

```

Figure 15.16: Generating a grid of cards in the Memory Game.

```
private class Card {
    public int status;
    public int value;

    public Card(int value) {
        this.status = 0;
        this.value = value;
    }

    public XML makeCard(int index) {
        switch(status) {
            case 0:
                return [[ <input type="image" alt="card"
                           src=[image] name=[index] /> ]]
                    <[ image = back_image, index = index ];
            case 1:
                return [[ <img src=[image] alt=[num] /> ]]
                    <[ image = images[value], num = value ];
            case 2:
                return [[ <img src=[image] alt="" /> ]]
                    <[ image = blank_image ];
            default:
                return null;
        }
    }
}
```

Figure 15.17: Representing a card in the Memory Game.

without errors, we know that no **JWIG** exceptions will be thrown while the game is being played. In particular, we are guaranteed that all documents being shown are valid XHTML according to the strict standard imposed by the DSD2 schema.

The **JWIG** runtime system, which is also used in the <bigwig> project, is tailor-made for session-based Web services. Each session thread is associated a unique URL which refers to a file on the server. This file at all times contains the most recent page shown to the client. The session code runs as a JVM thread that lives for the entire duration of the session. In contrast, sessions in Servlet and JSP services run as short-lived threads where the session identity is encoded using cookies or hidden input fields, as described in Section 15.1.3. This precludes sessions from being bookmarked, such that the client cannot suspend and later resume a session, and the history buffer in the browser typically gets cluttered with references to obsolete pages. In our solution, the session URL functions as an identity of the session, which avoids all these problems. These aspects are described in more detail in [40].

If we introduce an error in the program, by forgetting the name attribute in the input field in the welcome template, then the **JWIG** analyzer produces the following output:

```
*** Field 'howmany' is never available on line 68
*** Invalid XHTML at line 49
--- element 'input': requirement not satisfied:
<or xmlns="http://www.brics.dk/DSD/2.0/error">
  <attribute name="type">
    <union>
      <string value="submit" />
      <string value="reset" />
    </union>
  </attribute>
  <attribute name="name" />
</or>
```

In the first line, the receive analysis complains that the *howmany* field is never available from the client. The remainder of the error message is from the show analysis, which notices that the input element violates the quoted constraint from the XHTML schema. This particular validity error is not caught by DTD validation of the generated document. If the involved element contained gaps, the error message would include a print of all relevant template and string edges and values of the gap presence map, which shows the relevant plug operations. Clearly, such diagnostics are useful for debugging.

15.6.2 Performance

The **JWIG** implementation may be evaluated with respect to compile-time, analysis-time, or run-time. The compile-time performance is not an issue, since **JWIG** programs are simply piped through a JFlex desugarer and compiled using a standard Java compiler. The **JWIG** runtime performance is not particularly interesting, since we reuse the <bigwig> runtime system and the standard J2SE JVM. The critical component in our system is the extensive collection of static analyses that we perform on the generated class files.

Name	Lines	Templates	Shows/Exits	Total Time
Chat	80	4	3	5.370
Guess	94	8	7	7.147
Calendar	133	6	2	7.029
Memory	167	9	6	9.718
TempMan	238	13	3	7.719
WebBoard	766	32	24	9.769
Bachelor	1078	88	14	115.641
Jaoo	3923	198	9	35.997

Figure 15.18: The benchmark services.

Name	Load	Construct	Size Before	Simplify	Size After
Chat	3.205	1.871	238/388	0.232	107/99
Guess	3.242	1.993	286/442	0.152	113/89
Calendar	3.276	2.241	386/672	0.254	124/127
Memory	3.284	2.227	451/765	0.292	143/127
TempMan	3.290	2.604	779/1437	0.883	200/192
WebBoard	3.244	2.660	878/1285	0.775	422/287
Bachelor	3.255	4.303	2278/3676	21.862	1059/914
Jaoo	3.557	5.647	3009/4407	14.045	1406/1008

Figure 15.19: Flow graph construction.

As shown in Figure 15.7, the static analysis is a combination of many components, which we in the following quantify separately. Our benchmark suite, shown in Figure 15.18, is a collection of small to medium sized **JWIG** services, most of which have been converted from corresponding <bigwig> applications [39]. The right-most column shows the total time in seconds for the entire suite of program analyses. For all benchmarks, at most 150 MB memory is used.

The four larger ones are an XML template manager where templates can be uploaded and edited (TempMan), an interactive Web board for on-line discussions (WebBoard), a system for study administration (Bachelor), and a system for management of the JAoo 2001 conference (Jaoo).

Figure 15.19 shows the resources involved in computing the flow graphs on a 1 GHz Pentium III with 1 GB RAM running Linux. For each benchmark we show the time in seconds used by Soot, the time in seconds used by phases 1 through 7 described in Section 15.3, the size of the resulting flow graph, the time in seconds used by the simplifying Phase 8, and the size of the simplified flow graph. The flow graph sizes are shown as number of nodes and number of flow edges. The loading time is dominated by initialization of Soot. Phases 1 through 7 of the flow graph construction is seen to be linear in the program size. The time for the simplification phase strongly depends on the complexity of the document constructions, which explains the relatively large number for the Jaoo service. For all benchmarks, the simplification phase substantially reduces the flow graph size. Furthermore, recall that before the simplification phase, flow edges may have multiple variables, while after simplification, they all have exactly

Name	Time	Largest Size
Chat	0.103	2/1/5
Guess	0.105	2/1/3
Calendar	0.440	5/9/5
Memory	2.627	7/8/5
TempMan	0.203	11/13/9
WebBoard	1.189	9/11/11
Bachelor	24.673	47/83/24
Jaoo	5.067	33/45/48

Figure 15.20: Summary graph construction.

Name	Plug	Receive	Show	False Errors
Chat	0.002	0.004	0.953	0
Guess	0.015	0.002	1.638	0
Calendar	0.017	0.000	0.801	0
Memory	0.003	0.002	1.283	0
TempMan	0.004	0.015	0.720	0
WebBoard	0.016	0.003	1.882	0
Bachelor	0.133	0.009	61.406	0
Jaoo	0.059	0.002	7.620	0

Figure 15.21: Summary graph analysis.

one variable.

Figure 15.20 quantifies the computation of summary graphs, including the string analysis. For each benchmark we show the total time in seconds and the size of the largest summary graph, in terms of nodes, template edges, and non-trivial string edges. The relatively large numbers for the *Bachelor* example correctly reflects that it constructs complicated documents. Without graph simplification, the total time for the *Memory* example blows up to more than 15 minutes, while the *Jaoo* example was aborted after 90 minutes. We conclude that summary graph construction appears to be inexpensive in practice and that graph simplification is worth the effort.

Figure 15.21 deals with the subsequent analysis of all the computed summary graphs. For each benchmark we show the total time in seconds for each of the three analyses and the total number of false errors generated by the conservative analyses. In all cases, the times are smaller than for the other phases. Many benchmarks generated errors, but by careful inspection, they were all seen to correctly identify actual XHTML validity errors. Thus, the analysis appears to be precise enough to serve as a real help to the programmer.

We conclude that the **JWIG** prototype implementation is certainly feasible to use, but that there is room for performance improvements for the implementation.

15.7 Plans and Ideas for Future Development

Our current system can be extended and improved in many different directions which we plan to investigate in future work. These can be divided into language design, program analysis, and implementation issues, and are briefly described in the following.

15.7.1 Language Design

So far, the design of **JWIG** has focused on two topics that are central to the development of interactive Web services: sessions and dynamic construction of Web documents. However, there are many more topics that could benefit from high-level language-based solutions, as shown in [40].

One example is validation of form input. Many Web services apply intricate JavaScript client-side code in the Web documents for checking that input forms are filled in consistently. For instance, it is typical that certain fields must contain numbers or email addresses, or that some fields are optional depending on values entered in other fields. Proper error messages need to be generated when errors are detected such that the clients have the chance to correct them, and extra checks need to be performed on the server. The PowerForms language [37] has been developed to attack the problem of specifying form input validation requirements in a more simple and maintainable way based on regular expressions and boolean logic. It should be straightforward to integrate PowerForms into **JWIG**.

The current XML cast operation in **JWIG** is somewhat unsatisfactory for two reasons: 1) if a cast fails due to invalid XHTML, an exception is not thrown immediately since it is not detected until a subsequent show operation; and 2) its expressiveness is limited—for instance, unions of templates cannot be expressed. One solution to this may be to use DSD2 descriptions instead of constant templates in the cast operations. However, to generalize the analyses correspondingly, a technique for transforming a DSD2 description of an XML language into a summary graph is needed. We believe that this is theoretically possible—further investigation will show whether it is also practically feasible.

Another idea is to broaden the view from interactive Web services to whole Web sites comprising many services and documents. The Strudel system [86] has been designed to support generation and maintenance of Web sites according to the design principle that the underlying data, the site structure, and the visual presentation should be separated. A notion of data graphs allows the underlying data to be described, a specialized query language is used for defining the site structure, and an HTML template language that resembles the XML template mechanism in **JWIG** defines the presentation. We believe that the development of interactive services can be integrated into such a process. For sites that comprise both complex interactive session-based services and more static individual pages, the concepts in the `Service.Session` and `Service.Page` classes could be applied. **JWIG** could also benefit from a mechanism for specifying dependencies between the pages or sessions and the data, for instance, such that pages are automatically cached and only recomputed when certain variables or databases are modified.

We have shown that our template mechanism is suitable for constructing XHTML documents intended for presentation. If the underlying data of a Web service is rep-

resented with XML, as suggested by Strudel, we will need a general mechanism for extracting and transforming XML values. Currently, we only provide the plug operation for combining XML templates—a converse “unplug” operation would be required for deconstructing XML values. Preliminary results suggest that our notion of summary graphs and our analyses generalize to such general XML transformations [58]. XDuce [110] is a related research language designed to make type-safe XML transformations. In XDuce, types are simplified DTDs where we instead use the more powerful DSD2 notation. Furthermore, XDuce is a tiny functional language while **JWIG** contains the entire Java language. Instead of relying on a type system for ensuring that the various XML values are valid according to some schema definition, we perform data-flow analyses based on summary graphs. Based on these ideas, a current project aims to make **JWIG** a general and safe XML transformation language.

15.7.2 Program Analysis

The experiments indicate that the notion of summary graphs is suitable for modeling the XML template mechanism and that the analysis precision is adequate. However, the preliminary string analysis described in Section 15.4.1 can be improved. The modular design of the analyses makes it possible to replace this simple string analysis by a more precise one. For example, string concatenation operations could be modeled more precisely by exploiting the fact that regular languages are closed under finite concatenation. Because of loops in the flow graphs, this would in general produce context-free languages so a suitable technique for approximating these by regular languages is needed. That essentially amounts to applying widening for ensuring termination. Other operations, such as the substring methods, could also easily be modeled more precisely than with `anystring`. An advanced version of such an analysis would apply flow-sensitivity, such that e.g. `if` statements that branch according to the value of a string variable would be taken into account, and instead of modeling `receive` by `anystring`, the regular languages provided by PowerForms specifications could be applied. A natural extension to these ideas would be to add a “regular expression cast” operator to the **JWIG** language. As the other cast operations, that would provide a back-door to the analysis which occasionally can be convenient no matter how precise the analysis may be.

The current program analysis is based on the assumption that the medium used for communication with the clients is XHTML. However, since the show analysis is parameterized by a DSD2 description, validity with respect to any XML language describable by DSD2 can be checked instead. Two obvious alternatives are WML, *Wireless Markup Language* [217], which is used for mobile WAP devices with limited network bandwidth and display capabilities, and VoiceXML, *Voice Extensible Markup Language* [33], for audio-based dialogs. Such languages can be described precisely with DSD2. Only the receive analysis requires modification since it needs to identify the forms and fields, or whatever the equivalent notions are in other formalisms.

15.7.3 Implementation

Our current implementation is a prototype developed to experiment with the design and test the performance. This means that there are plenty of ways to improve the

performance of the analysis and the runtime system.

We plan to apply the metafront syntax macros [42] in a future version to improve the quality of the parsing error messages. This will also allow us to experiment with syntax macros as a means for developing highly domain-specific languages in the context of Java-based interactive Web services.

The application of Soot in the generation of flow graphs is sometimes a bottleneck in the analysis, even though the theoretical complexity of this translation is trivial compared to the other analysis phases. This suggests that we use a tailor-made byte-code to flow-graph converter instead.

Finally, we believe that it is possible to significantly improve the runtime performance for **JWIG** services by integrating the **JWIG** runtime system with a Java Enterprise Edition server. For instance, this allows `Service.Page` to become essentially as efficient as JSP code by exploiting that the threads are never suspended by `show` statements. JRockit [4] is a commercial JVM implementation which is tuned for Web servers with high loads. In particular, it supports light-weight threads which will significantly reduce the overhead induced by our session model.

15.8 Conclusion

We have defined **JWIG** as an extension of the Java language with explicit high-level support for two central aspects of interactive Web services: 1) sessions consisting of sequences of client interactions and 2) dynamic construction of Web pages. Compared to other Web service programming languages, these extensions can improve the structure of the service code. In addition to being convenient during development and maintenance of Web services, this allows us to perform specialized program analyses that check at compile time whether or not runtime errors may occur due to the construction of Web pages or communication with the clients via input forms. The program analyses are based on a unique notion of *summary graphs* which model the flow of document fragments and text strings through the program. These summary graphs prove to contain exactly the information needed to provide all the desired static guarantees of the program behavior.

This article can be viewed as a case study in program analysis. It contains a total of seven analyses operating on different abstractions of the source program: one for making receive edges during flow graph construction, the reaching definitions analysis in the flow-graph simplification phase, the string analysis, the summary graph construction, and the plug, receive, and show analyses. The whole suite of analyses is modular in the sense that each of them easily can be replaced by a more precise or efficient one, if the need should arise. If, for example, future experience shows that the control-flow information in the flow graphs is too imprecise, one could apply a *variable-type analysis* [203] instead of CHA. Or, if the string analysis should turn out to be inadequate for developing, e.g., WML services, it could be replaced by another. Analysis correctness is given by the correctness of each phase. For instance, the flow graphs conservatively approximate the behavior of the original **JWIG** programs, the summary graphs conservatively model the template constructions with respect to the flow graphs, and the validity results given by the show analysis are conservative with respect to the summary graphs.

The language extensions permit efficient implementation, and despite the theoretical worst-case complexities of the program analyses, they are sufficiently precise and fast for practical use.

All source code for our **JWIG** implementation, including API specifications and the DSD2 schema for XHTML 1.0, is available from <http://www.brics.dk/JWIG/>.

Chapter 16

Static Analysis for Dynamic XML

with Aske Simon Christensen and Michael I. Schwartzbach

Abstract

We describe the *summary graph* lattice for dataflow analysis of programs that dynamically construct XML documents. Summary graphs have successfully been used to provide static guarantees in the **JWIG** language for programming interactive Web services. In particular, the **JWIG** compiler is able to check validity of dynamically generated XHTML documents and to type check dynamic form data. In this paper we present summary graphs and indicate their applicability for various scenarios. We also show that summary graphs have exactly the same expressive power as the regular expression types from XDuce, but that the extra structure in summary graphs makes them more suitable for certain program analyses.

16.1 Introduction

XML documents will often be generated dynamically by programs. A common example is XHTML documents being generated by interactive Web services in response to requests from clients. Typically, there are no static guarantees that the generated documents are valid according to the DTD for XHTML. In fact, a quick study of the outputs from many large commercial Web services shows that most generated documents are in fact invalid. This is not a huge problem, since the browsers interpreting this output are quite forgiving and do a fair job of rendering invalid documents. Increasingly, however, Web services will generate output in other XML languages for less tolerant clients, many of whom will themselves be Web services.

Thus it is certainly an interesting question to statically guarantee validity of dynamically generated XML. Our approach is to perform a dataflow analysis of the program generating XML documents. This is a standard technique that is basically just parameterized by the finite lattice used to abstract the computed values and the transfer functions modeling the statements. The contribution described in this paper is the definition of an appropriate lattice of *summary graphs* that strikes a balance between expressive power and complexity. We show how summary graphs have been used to

efficiently analyze realistic Web services with great accuracy. Also, we discuss what kind of operations on XML values that successfully can be captured by such dataflow analysis. Finally, we show that summary graphs have the same expressive power as the regular expression types of XDuce [110, 112, 111].

16.2 XML Templates

We have concretely analyzed programs in the **JWIG** language, which is an extension of Java designed for programming interactive Web services. **JWIG** is a descendant of the **<bigwig>** language [40]. For the current discussions, we only need to consider how XML documents are built. **JWIG** is based on the notion of XML *templates*, which are just sequences of XML trees containing named *gaps*. A special *plug* operation is used to construct new templates by inserting existing templates or strings into gaps in other templates. Using XHTML as an example, the main method of a **JWIG** program manipulating templates could look like:

```
public void main() {
    XML wrapper = [[ <html>
                        <head>
                            <title>JWIG Example</title>
                        </head>
                        <body>
                            <[contents]>
                        </body>
                    </html> ]];

    XML item = [[ <li> <[text]> </li> <[items]> ]];

    XML x = [[ <ul class=[kind]> <[items]> </ul> ]];

    for (int i=0; i<n; i++) {
        x = x<[items = item<[text=i]]>
    }
    show wrapper<[contents=x]<[kind="large"]>
}
```

Gaps appear either as *template gaps*, such as `contents`, or as *attribute gaps*, such as `kind`. Both strings and templates may be plugged into template gaps, whereas attribute gaps only allow strings. The plug operation, $x<[g=y]$, returns a copy of x where copies of y have been inserted into all g gap. Template constants are denoted by `[[...]]`.

Note that XML values need not be constructed bottom-up, since gaps can be left in templates as targets for later plug operations. Also, a plug operation will fill in all occurrences of the given gap, even if they originate from different subtemplates. The more common language design of building XML values from constructors is a special case of this mechanism, since e.g. a construction like `1[X]` from XDuce corresponds to `[[<1><[g]></1>]]` `<[g=X]`. The plug operation has proved itself to be flexible and intuitive. Also, it is convenient to write larger constant fragments in ordinary XML syntax rather than using nested constructor invocations.

16.3 Summary Graphs

We want to perform dataflow analysis of programs constructing XML values by plugging together templates. This key ingredient for such an analysis is a finite lattice for summarizing the state of a computation for each program point. Based on earlier experiences [195, 39], we have defined the lattice of *summary graphs*. Such a graph has as nodes the set of template constants occurring in the given program. The edges correspond to possible pluggings of gaps with strings or other templates. Given a concrete program, we let G be the set of gap names that occur and N be a set of *template indices* denoting the instances of XML template constants. A summary graph SG is formally defined as follows:

$$SG = (R, T, S, P)$$

where:

- $R \subseteq N$ is a set of *root nodes*,
- $T \subseteq N \times G \times N$ is a set of *template edges*,
- $S : N \times G \rightarrow REG$ is a *string edge* map, and
- $P : G \rightarrow 2^N \times \Gamma \times \Gamma$ is a *gap presence* map.

Here $\Gamma = 2^{\{OPEN, CLOSED\}}$ is the *gap presence lattice* whose ordering is set inclusion, and REG is the set of regular languages over the Unicode alphabet.

Intuitively, the language $L(SG)$ of a summary graph SG is the set of XML documents that can be obtained by unfolding its templates, starting from a root node and plugging templates and strings into gaps according to the edges. The presence of a template edge $(n_1, g, n_2) \in T$ informally means that the template with index n_2 may be plugged into the g gaps in the template with index n_1 , and a string edge $S(n, g) = L$ means that every string in the regular language L may be plugged into the g gaps in the template with index n .

The gap presence map, P , specifies for each gap name g which template constants may contain open g gaps reachable from a root and whether g gaps may or must appear somewhere in the unfolding of the graph, either as template gaps or as attribute gaps. The first component of $P(g)$ denotes the set of template constants with open g gaps, and the second and third components describe the presence of template gaps and attribute gaps, respectively. Given such a triple, $P(g)$, we let $nodes(P(g))$ denote the first component. For the other components, the value OPEN means that the gaps may be open, and CLOSED means that they may be closed or never have occurred. At runtime, if a document is shown with open template gaps, these are treated as empty strings. For open attribute gaps, the entire attribute is removed. We need the gap presence information in the summary graphs to 1) determine where edges should be added when modeling plug operations, 2) model the removal of gaps that remain open when a document is shown, and 3) detect that plug operations may fail because the specified gaps have already been closed.

This unfolding of summary graphs is explained more precisely with the following formalization:

$$unfold(SG) = \{d \mid \exists r \in R : SG, r \vdash t(r) \Rightarrow d \text{ where } SG = (R, T, S, P)\}$$

Here, $t(n)$ denotes the template with index n . The *unfolding relation*, \Rightarrow , is defined by induction in the structure of the XML template. For the parts that do not involve gaps the definition is a simple recursive traversal:

$$\begin{array}{c}
\overline{SG, n \vdash str \Rightarrow str} \\
\\
\frac{SG, n \vdash xml_1 \Rightarrow xml'_1 \quad SG, n \vdash xml_2 \Rightarrow xml'_2}{SG, n \vdash xml_1 xml_2 \Rightarrow xml'_1 xml'_2} \\
\\
\frac{SG, n \vdash atts \Rightarrow atts' \quad SG, n \vdash xml \Rightarrow xml'}{SG, n \vdash \langle name \ atts \rangle xml \ \langle /name \rangle \Rightarrow \langle name \ atts' \rangle xml' \ \langle /name \rangle} \\
\\
\overline{SG, n \vdash \varepsilon \Rightarrow \varepsilon} \\
\\
\overline{SG, n \vdash name = "str" \Rightarrow name = "str"} \\
\\
\frac{SG, n \vdash atts_1 \Rightarrow atts'_1 \quad SG, n \vdash atts_2 \Rightarrow atts'_2}{SG, n \vdash atts_1 \ atts_2 \Rightarrow atts'_1 \ atts'_2}
\end{array}$$

For template gaps we unfold according to the string edges and template edges and check whether the gap may be open:

$$\begin{array}{c}
\frac{str \in S(n, g)}{(R, T, S, P), n \vdash \langle [g] \rangle \Rightarrow str} \\
\\
\frac{(n, g, m) \in T \quad (R, T, S, P), m \vdash t(m) \Rightarrow xml}{(R, T, S, P), n \vdash \langle [g] \rangle \Rightarrow xml} \\
\\
\frac{n \in nodes(P(g))}{(R, T, S, P), n \vdash \langle [g] \rangle \Rightarrow \langle [g] \rangle}
\end{array}$$

For attribute gaps we unfold according to the string edges, and check whether the gap may be open:

$$\begin{array}{c}
\frac{str \in S(n, g)}{(R, T, S, P), n \vdash name = [g] \Rightarrow name = "str"} \\
\\
\frac{n \in nodes(P(g))}{(R, T, S, P), n \vdash name = [g] \Rightarrow name = [g]}
\end{array}$$

Using a function *close* that removes all remaining gaps in an XML template, we now define the language of a summary graph by:

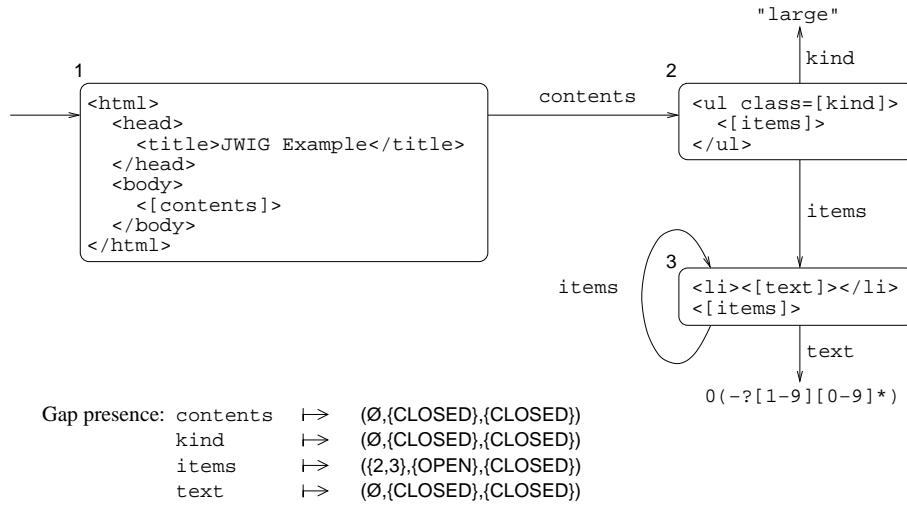
$$L(SG) = \{close(d) \mid d \in unfold(SG)\}$$

Summary graphs for a given program form a lattice where the ordering is defined as one would expect:

$$\begin{array}{c}
(R_1, T_1, S_1, P_1) \sqsubseteq (R_2, T_2, S_2, P_2) \Leftrightarrow \\
R_1 \subseteq R_2 \wedge T_1 \subseteq T_2 \wedge \forall n \in N, g \in G : S_1(n, g) \subseteq S_2(n, g) \wedge P_1(g) \sqsubseteq P_2(g)
\end{array}$$

where the ordering on gap presence maps is defined by componentwise set inclusion. This respects language inclusion: if $SG_1 \sqsubseteq SG_2$, then $L(SG_1) \subseteq L(SG_2)$, but the converse implication is false.

Continuing the previous example, the summary graph inferred for the XML value being shown to the client is:



It is relatively simple to define appropriate transfer functions and perform a standard monotone dataflow analysis for the **JWIG** language [57]. The most interesting example is the template plug operation, $z = x <[g = y]$ where y is of type XML. This operation assigns to z a copy of x where y has been plugged into all g gaps. It is modeled by the following transfer function:

$$\begin{aligned}
 (R_z, T_z, S_z, P_z) = & \\
 & (R_x, \\
 & T_x \cup T_y \cup \{(n, g, m) \mid n \in \text{nodes}(P_x(g)) \wedge m \in R_y\}, \\
 & \lambda(m, h).S_x(m, h) \cup S_y(m, h), \\
 & \lambda h. \text{if } h = g \text{ then } P_y(h) \\
 & \quad \text{else } (p_x \cup p_y, \text{merge}(t_x, t_y), \text{merge}(a_x, a_y)) \\
 & \quad \text{where } P_x(h) = (p_x, t_x, a_x) \text{ and } P_y(h) = (p_y, t_y, a_y))
 \end{aligned}$$

where

$$\text{merge}(\gamma_1, \gamma_2) = \text{if } \gamma_1 = \{\text{OPEN}\} \vee \gamma_2 = \{\text{OPEN}\} \text{ then } \{\text{OPEN}\} \text{ else } \gamma_1 \cup \gamma_2.$$

The tuples (R_x, T_x, S_x, P_x) and (R_y, T_y, S_y, P_y) denote the summary graphs that are associated to x and y at the entry point, and (R_z, T_z, S_z, P_z) is the summary graph for z at the exit point. The roots in the resulting graph are those of the x graph since it represents the outermost template. The template edges become the union of those in the two given graphs plus a new edge from each node that may have open gaps of the given name to each root in the second graph. The string edge sets are simply joined without adding new information. For the gaps that are plugged into, we take the gap presence information from the second graph. For the other gaps we merge the information appropriately.

The set of regular expressions describing computed string values are determined through a separate dataflow analysis. Thus we obtain summary graphs that conservatively describe all computed XML values at each program point; for more details see [57]. The worst-case complexity for this algorithm is $O(n^6)$, where n is the size of the program.

16.4 Static Guarantees in JWIG

Based on the inferred summary graphs, various static guarantees can be issued.

First we must deal with a self-inflicted problem stemming from the liberal gap-and-plug mechanism. We need to know that whenever a gap is being plugged, it is actually present in the XML value. However, this information is directly available in the gap presence map component, and a trivial inspection suffices. Note that in the special case of constructors, corresponding to $[[\text{<1><[g]></1> }]]$ <[g=X] , this property trivially holds.

Second, we need to validate the XML values being generated. This is dependent of the XML language in question, which must first be specified. We could use ordinary DTDs for this purpose, but have instead chosen the XML schema language DSD2 [160], which is a further development of DSD [133]. We have a general algorithm that given a summary graph SG and a DSD2 schema can verify that every document in $L(SG)$ validates according to the schema. The DSD2 schema for XHTML is more comprehensive than most others, since it specifies correct formats for attribute values that are URIs and includes several context-sensitive requirements that are only stated as comments in the official DTD.

The final analysis is specific to XHTML and verifies that the form data expected by the server is actually present in the last document being shown to the client.

These analyses are fully specified in [57]. They have rather high worst-case complexities, but are in practice able to handle realistic program. The following table shows statistics for some small to largish benchmarks.

Name	Lines	Templates	Largest Graph	Total Time
Chat	80	4	(2,6)	5.37
Guess	94	8	(2,4)	7.15
Calendar	133	6	(5,14)	7.03
Memory	167	9	(7,13)	9.72
TempMan	238	13	(11,22)	7.72
WebBoard	766	32	(9,22)	9.77
Bachelor	1,078	88	(47,107)	115.64
Jaoo	3,923	198	(33,93)	36.00

To indicate the scale of each benchmark, we give the number of lines of code and the number of template constants. We also show the size of the largest summary graph computed for each benchmark by indicating the number of nodes (reachable from the roots) and the number of edges. The time, measured in seconds, is the total for inferring summary graphs and performing the three subsequent analyses.

16.5 Analyzing Deconstruction

The present version of the **JWIG** language does not contain any mechanism for deconstruction of XML values. However, the summary graph analysis can—with simple modifications—easily handle this.

We extend the **JWIG** language with a notion of deconstruction based on XPath [61]

that generalizes most other proposals. Since we are working on XML values containing gaps, we get two variations.

The *select* expression looks like $x > [path]$, where x is an XML value and $path$ is a location path. The result is an array of XML values corresponding to those subtrees that are rooted by the elements of the computed node set. The XML value x is first closed, that is, all gaps are removed, as was the case before *show* operations. Other deconstruction mechanisms can clearly be obtained as special cases. For example, the pattern matching of XDuce corresponds to writing a selector path for each case and trying them out in turn.

By exploiting the gap mechanism, we can also introduce a complementary operation that replaces parts of an XML values by gaps. The *gapify* expression looks like $x > [path=g]$, where x is an XML value, $path$ is a location path, and g is an identifier. The result is a copy of x where the subtrees rooted by the computed node set are replaced by gaps named g . Again, x is first closed. If x is the XML value:

```
<html>
  <head>
    <title>
      Jwig example
    </title>
  </head>
  <body>
    <ul class="large">
      <li>0</li>
      <li>1</li>
      <li>2</li>
      <li>3</li>
    </ul>
  </body>
</html>
```

then the result of $x > [//li [text() > '0']]$ is the following XML array with three entries:

```
{ [[ <li>1</li> ]], [[ <li>2</li> ]], [[ <li>3</li> ]] }
```

and the result of $x > [//li [text() > '0' = g]]$ is the “negative image” in form of the XML value with gaps named g in place of the selected subtemplates:

```
<html>
  <head>
    <title>
      Jwig example
    </title>
  </head>
  <body>
    <ul class="large">
      <li>0</li>
      <[g]>
      <[g]>
      <[g]>
    </ul>
```

```

    </body>
  </html>

```

The summary graph analysis can be extended with transfer functions for select and gapify. In [57], the close operation only occurs in connection with show operations. However, because of our extensions with the select and gapify we now need to model close operations separately. We must remove all gaps that might be open according to the gap presence map. To model that a template gap is removed, one simply adds a template edge to a node with an empty template. For attribute gaps, we need a small modification of the string edge component of the summary graph structure:

$$S : N \times G \rightarrow REG \times 2^{\{\div\}}$$

The new element \div represents the possibility that the designated attribute might be removed. The definition of the unfolding relation is extended with a rule describing this meaning:

$$\frac{(-, \div) \in S(n, g)}{(R, T, S, P), n \vdash name = [g] \Rightarrow \varepsilon}$$

To model that an attribute gap is removed in a close operation, we just add \div to the appropriate string edge. The gap presence map of the result of a close operation maps all gaps to $(\emptyset, \{\text{CLOSED}\}, \{\text{CLOSED}\})$.

The core observation when modeling select and gapify operations is in both cases that an XPath selector path can be evaluated symbolically on a summary graph. The resulting node set is represented abstractly by assigning a status to each element in all templates assigned to nodes in the summary graph. The possible status values are:

- *all*: every occurrence of this element belongs to the node set in every unfolding of the summary graph;
- *some*: at least one occurrence of this element belongs to the node set in every unfolding of the summary graph;
- *definite*: the conditions for both *all* and *some* are satisfied;
- *none*: no occurrences of this element belong to the node set in any unfolding of the summary graph;
- *don't know*: none of the above can be determined.

This forms a 5-valued logic reminiscent of the logic used when analyzing validity with respect to DSD2 schemas [57]. Based on these status values, it is straightforward conservatively to compute summary graphs for the results of select and gapify.

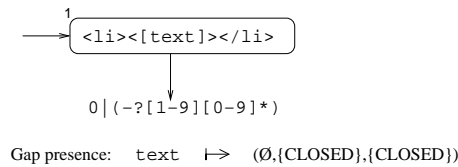
For a select expression, all subtemplates whose root elements do not have status *none* are added to the summary graph, inherit all relevant edges, and are made the only root nodes.

For a gapify expression, all subtemplates whose root elements do not have status *none* are replaced by a gap named g . If the status is *some* or *don't know*, the new gap will have a template edge to a copy of the old subtemplate. The gap presence map of the new summary graph will be

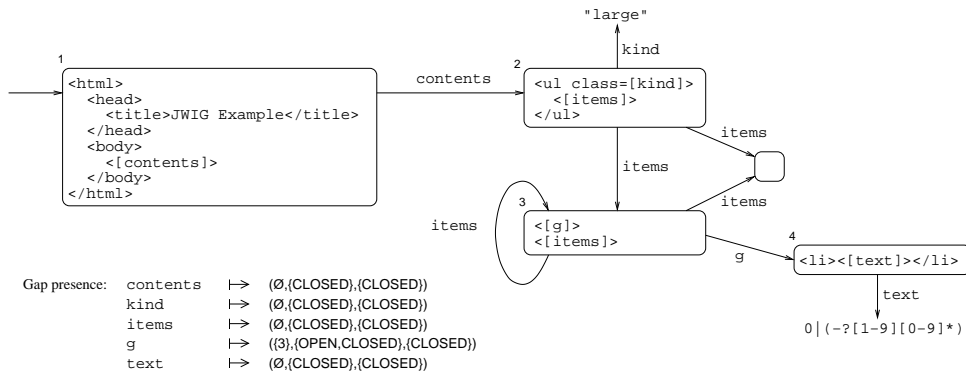
$$\lambda h. \text{if } h = g \text{ then } (hits, any, \{\text{CLOSED}\}) \text{ else } (\emptyset, \{\text{CLOSED}\}, \{\text{CLOSED}\})$$

where *hits* is the set of all template nodes containing an element with status different from *none*, and *any* is {OPEN} if there is an element with status *definite* or *some*, {CLOSED} if all elements have status *none*, and {OPEN, CLOSED} otherwise.

Continuing the example from Section 16.4, the result of the select expression is described by



and the result of the gapify expression by



We are currently implementing the select and gapify operations and the associated extensions of the summary graph analysis in the **JWIG** system to test the analysis precision and performance in practice.

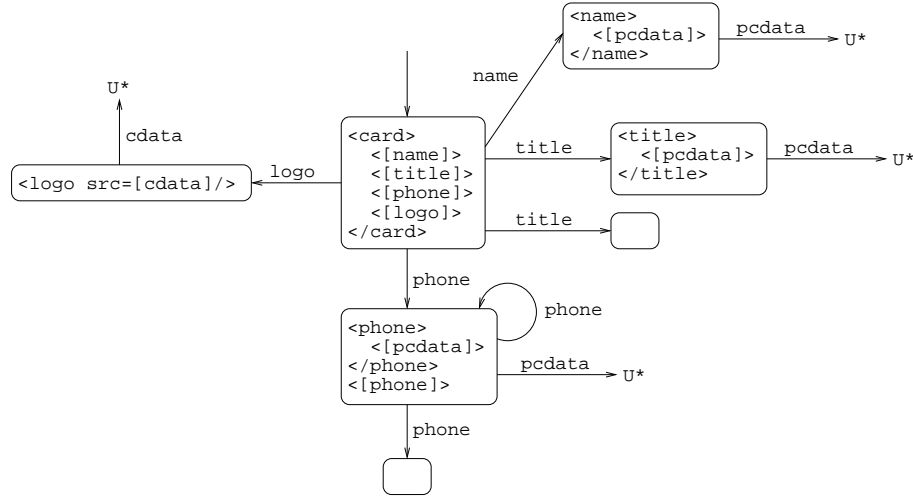
Deconstruction is mainly relevant for XML values that are imported from external sources. To obtain non-trivial analyses, we need to obtain summary graph descriptions of such values. In practice this will be done by performing automatic translations from DTDs or DSD2 schemas. We believe that such translations can be made sufficiently precise. As an example, consider the following DTD:

```

<!ELEMENT card (name,title?,phone+,logo)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT logo EMPTY>
<!-- ATTLIST logo src CDATA #REQUIRED -->

```

It is exactly captured by the following summary graph:



where all gaps are closed and U is the set of all Unicode characters. For a richer schema language as DSD2 the translation will of course become more complex, and it will in some cases be necessary to perform conservative approximations.

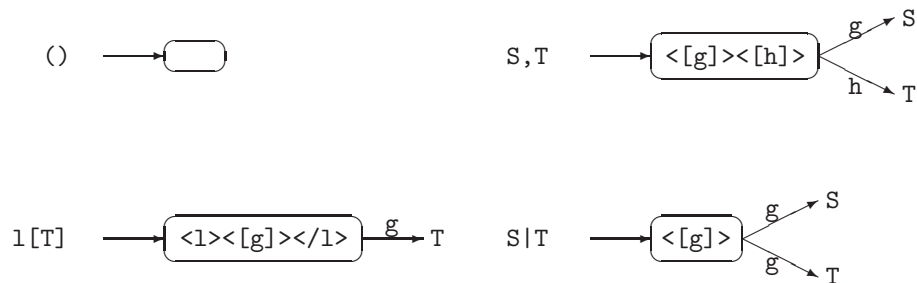
16.6 Regular Expression Types

Summary graphs turn out to have the same expressive power as the regular expression types of XDuce [110]. To be exact, this comparison only holds for a restricted version of summary graphs. Since XDuce does not support attributes, those must be left out. Also, summary graphs allow for restrictions on character data appearing in element contents, which is also not supported by XDuce.

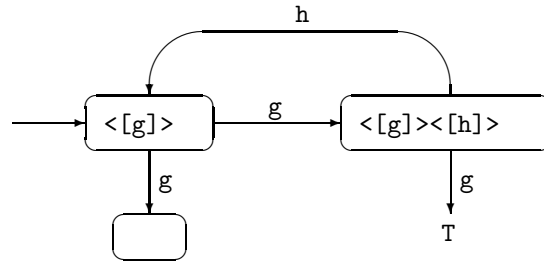
Regular expression types are essentially solutions to recursive equations using the operators $()$ (the empty value), $1[T]$ (singleton element), $S|T$ (union), and S,T (sequencing). For example, the derived operator T^* is defined by the equation:

$$X = T, X \mid ()$$

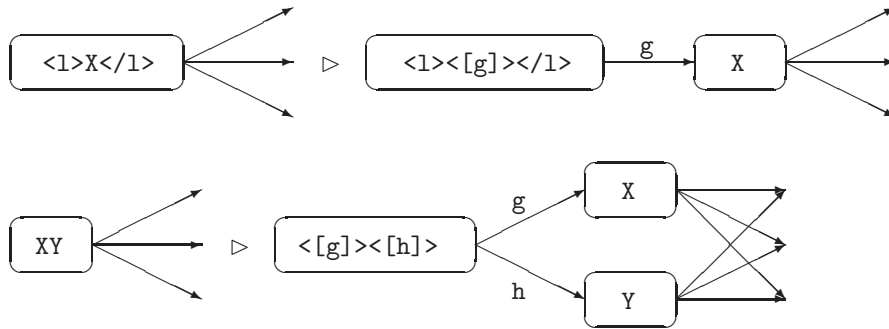
A regular expression type defines a set of XML values corresponding to all finite unfoldings. It is now a simple matter to build inductively a summary graph that defines the same set of XML values. The four operators are modeled by summary graphs as follows:



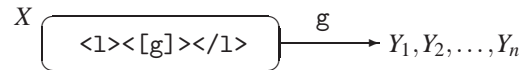
All gaps are closed in these summary graphs. An edge to a variable is modeled by an edge to the root node of the summary graph corresponding to its right-hand side. For example, the derived summary graph for T^* is:



Note that the sequencing operator is associative as required in XDuce. The inverse translation is equally straightforward, but requires that the summary graph is first *normalized*. First, all open gaps are translated into closed ones by adding a template edge to an empty template. Second, all non-empty template constants are decomposed into one of the forms $\langle 1 \rangle \langle [g] \rangle \langle /1 \rangle$ or $\langle [g] \rangle \langle [h] \rangle$. This is done by repeatedly applying the rewritings sketched by:



Given a normalized summary graph, we first assign a type variable to each node. Then for each node of the form:

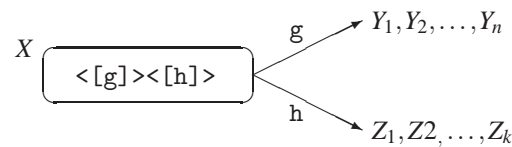


we define the type equations:

$$X = 1[Y]$$

$$Y = Y_1 \mid \dots \mid Y_n$$

for each node of the form:



we define the type equations:

$$X = Y, Z$$

$$Y = Y_1 \mid \dots \mid Y_n$$

$$Z = Z_1 \mid \dots \mid Z_k$$

and for empty nodes:



we define the type equation:

$$X = ()$$

Finally, for the root nodes R_1, \dots, R_n we define the type equation:

$$R = R_1 \mid \dots \mid R_n$$

and the type R is the final result of the translation. These two translations demonstrate the close relationship between our approach and that of XDuce.

Our analysis is thus also able to infer regular expression types for programs that dynamically construct XML values. This result does not directly apply to XDuce, since we infer different types for variables at each program point. It is, however, possible to subsequently verify a subtype relationship between each declared and inferred type. The resulting type reconstruction algorithm is of course not complete with respect to the type rules of XDuce [110]; in fact, their abilities to accept programs are most likely incomparable.

Note that the summary graph lattice contains extra structure in the form of the root sets and the gap presence maps. Also, the lattice order is more fine-grained than the language inclusion used as subtyping in XDuce. All this constitutes a scaffolding that is required during analysis but is not needed to express the final results. Note also that the analysis uses a kind of constructor polyvariance, since constructors in the summary graph are represented once for each invocation site. We believe that similar ideas would need to be developed for XDuce, if flow-sensitive type reconstruction were to be attempted directly.

16.7 Conclusion

We have presented the lattice of summary graphs as a convenient means for abstracting sets of XML values during dataflow analyses of programs that dynamically construct XML documents. Summary graphs have been used in the fully implemented **JWIG** language, and we have indicated the applicability for other scenarios.

Bibliography

- [1] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers – Principles, Techniques, and Tools*, Addison-Wesley, November 1985.
- [2] L. ALSCHULER, *XML Schemas: Last word on last call*, July 2000.
<http://www.xml.com/pub/a/2000/07/05/specs/lastword.html>.
- [3] V. APPARAO ET AL., *Document Object Model (DOM) level 1 specification*, October 1998. W3C Recommendation.
<http://www.w3.org/TR/REC-DOM-Level-1/>.
- [4] APPEAL VIRTUAL MACHINES, *JRockit – the faster server JVM*, 2002.
<http://www.jrockit.com/>.
- [5] K. R. APT, *Ten years of Hoare’s logic: A survey—part I*, ACM Transactions on Programming Languages and Systems, 3 (1981), pp. 431–483.
- [6] K. ARNOLD, J. GOSLING, AND D. HOLMES, *The Java Programming Language*, Addison-Wesley, 3rd ed., June 2000.
- [7] D. ATKINS, T. BALL, M. BENEDIKT, G. BRUNS, K. COX, P. MATAGA, AND K. REHOR, *Experience with a domain specific language for form-based services*, in Proc. Conference on Domain-Specific Languages, DSL ’97, USENIX, October 1997.
- [8] D. ATKINS, T. BALL, G. BRUNS, AND K. COX, *Mawl: a domain-specific language for form-based services*, IEEE Transactions on Software Engineering, 25 (1999), pp. 334–346.
- [9] L. ATKINSON, *Core PHP Programming*, Prentice Hall, 2nd ed., August 2000.
- [10] A. AYARI AND D. BASIN, *Bounded model construction for monadic second-order logics*, in Proc. 12th International Conference on Computer-Aided Verification, CAV ’00, vol. 1855 of LNCS, Springer-Verlag, July 2000.
- [11] A. AYARI, D. BASIN, AND S. FRIEDRICH, *Structural and behavioral modeling with monadic logics*, in Proc. 29th International Symposium on Multiple-Valued Logic, ISMVL ’99, IEEE Computer Society, May 1999.
- [12] A. AYARI, D. BASIN, AND A. PODELSKI, *LISA: A specification language based on WS2S*, in Proc. 11th International Workshop on Computer Science Logic, CSL ’97, vol. 1414 of LNCS, Springer-Verlag, August 1997.

- [13] T. BALL AND S. K. RAJAMANI, *Bebop: A symbolic model checker for boolean programs*, in Proc. 7th International Workshop on SPIN Software Model Checking, SPIN '00, vol. 1885 of LNCS, Springer-Verlag, August/September 2000.
- [14] P. BARFORD, A. BESTAVROS, A. BRADLEY, AND M. CROVELLA, *Changes in web client access patterns: Characteristics and caching implications*, World Wide Web Journal, 2 (1999), pp. 15–28. Kluwer.
- [15] G. BARISH AND K. OBRACZKA, *World Wide Web caching: Trends and techniques*, IEEE Communications Magazine, Internet Technology Series, 38 (2000), pp. 178–184.
- [16] D. BASIN AND S. FRIEDRICH, *Combining WSIS and HOL*, in Frontiers of Combining Systems 2, D. M. Gabbay and M. de Rijke, eds., vol. 7 of Studies in Logic and Computation, Research Studies Press/Wiley, February 2000, pp. 39–56.
- [17] D. BASIN, S. FRIEDRICH, AND S. MÖDERSHEIM, *B2M: A semantic based tool for BLIF hardware descriptions*, in Proc. 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD '00, vol. 1954 of LNCS, Springer-Verlag, November 2000, pp. 91–107.
- [18] D. BASIN AND N. KLARLUND, *Automata based symbolic reasoning in hardware verification*, Formal Methods In System Design, 13 (1998), pp. 255–288. Kluwer. Earlier version in CAV '95, vol. 939 of LNCS, Springer-Verlag.
- [19] K. BAUKUS, K. STAHL, S. BENSALAM, AND Y. LAKHNECH, *Abstracting WSIS systems to verify parameterized networks*, in Proc. 6th International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '00, vol. 1785 of LNCS, Springer-Verlag, March/April 2000.
- [20] B. BEHLENDORF ET AL., *The Apache HTTP server project*, 2002. <http://httpd.apache.org/>.
- [21] M. BENEDIKT, T. REPS, AND M. SAGIV, *A decidable logic for describing linked data structures*, in Programming Languages and Systems, Proc. 8th European Symposium on Programming, ESOP '99, vol. 1576 of LNCS, Springer-Verlag, March 1999.
- [22] T. BERNERS-LEE, R. FIELDING, AND H. FRYSTYK, *Hypertext transfer protocol – HTTP/1.0*, May 1996. RFC1945. <http://www.w3.org/Protocols/rfc1945/rfc1945>.
- [23] M. BIEHL, N. KLARLUND, AND T. RAUHE, *Algorithms for guided tree automata*, in Proc. 1st International Workshop on Implementing Automata, WIA '96, vol. 1260 of LNCS, Springer-Verlag, August 1996.
- [24] M. BIEHL, N. KLARLUND, AND T. RAUHE, *Mona: Decidable arithmetic in practice*, in Proc. 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT '96, vol. 1135 of LNCS, Springer-Verlag, September 1996.

- [25] P. V. BIRON AND A. MALHOTRA, *XML Schema part 2: Datatypes*, May 2001. W3C Recommendation. <http://www.w3.org/TR/xmlschema-2/>.
- [26] P. E. BLACK AND P. J. WINDLEY, *Inference rules for programming languages with side effects in expressions*, in Proc. 9th International Conference on Theorem Proving in Higher Order Logics, TPHOLs '96, vol. 1125 of LNCS, Springer-Verlag, August 1996.
- [27] S. BOAG ET AL., *XQuery 1.0: An XML query language*, December 2001. W3C Working Draft. <http://www.w3.org/TR/xquery/>.
- [28] J.-P. BODEVEIX AND M. FILALI, *Quantifier elimination technics for program validation*, tech. rep., IRIT 97-44-R, 1997.
- [29] J.-P. BODEVEIX AND M. FILALI, *FMona: A tool for expressing validation techniques over infinite state systems*, in Proc. 6th International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '00, vol. 1785 of LNCS, Springer-Verlag, March/April 2000.
- [30] B. BOS, H. W. LIE, C. LILLEY, AND I. JACOBS, *Cascading style sheets, level 2, CSS2 specification*, May 1998. W3C Recommendation. <http://www.w3.org/TR/REC-CSS2/>.
- [31] R. BOURRET, *Namespace myths exploded*, March 2000. <http://www.xml.com/pub/a/2000/03/08/namespaces/>.
- [32] R. BOURRET, J. COWAN, I. MACHERIUS, AND S. S. LAURENT, *Document definition markup language (DDML) specification, version 1.0*, January 1999. W3C Note. <http://www.w3.org/TR/NOTE-ddml>.
- [33] L. BOYER, P. DANIELSEN, J. FERRANS, G. KARAM, D. LADD, B. LUCAS, AND K. REHOR, *Voice eXtensible Markup Language, version 1.0*, May 2000. W3C Note. <http://www.w3.org/TR/voicexml/>.
- [34] C. BRABRAND, *Synthesizing safety controllers for interactive Web services*, Master's thesis, Department of Computer Science, University of Aarhus, December 1998.
- [35] C. BRABRAND, *<bigwig> Version 1.3 – Reference Manual*, BRICS, Department of Computer Science, University of Aarhus, September 2000. Notes Series NS-00-4.
- [36] C. BRABRAND, A. MØLLER, S. OLESEN, AND M. I. SCHWARTZBACH, *Language-based caching of dynamically generated HTML*, World Wide Web Journal, (2002). Kluwer. (See Dissertation Chapter 12).
- [37] C. BRABRAND, A. MØLLER, M. RICKY, AND M. I. SCHWARTZBACH, *PowerForms: Declarative client-side form field validation*, World Wide Web Journal, 3 (2000), pp. 205–314. Kluwer. (See Dissertation Chapter 11).

- [38] C. BRABRAND, A. MØLLER, A. SANDHOLM, AND M. I. SCHWARTZBACH, *A runtime system for interactive Web services*, Computer Networks, 31 (1999), pp. 1391–1401. Elsevier. Also in Proc. 8th International World Wide Web Conference, WWW8. (See Dissertation Chapter 10).
- [39] C. BRABRAND, A. MØLLER, AND M. I. SCHWARTZBACH, *Static validation of dynamically generated HTML*, in Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01, June 2001, pp. 221–231. (See Dissertation Chapter 13).
- [40] C. BRABRAND, A. MØLLER, AND M. I. SCHWARTZBACH, *The <bigwig> project*, ACM Transactions on Internet Technology, 2 (2002), pp. 79–114. (See Dissertation Chapter 9).
- [41] C. BRABRAND AND M. I. SCHWARTZBACH, *Growing languages with metamorphic syntax macros*, in Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM '02, January 2002.
- [42] C. BRABRAND AND M. I. SCHWARTZBACH, *The metafront system: Extensible syntax processing*. In preparation, 2002.
- [43] T. BRAY, C. FRANKSTON, AND A. MALHOTRA, *Document content description for XML*, July 1998. W3C Note. <http://www.w3.org/TR/NOTE-dcd>.
- [44] T. BRAY, D. HOLLANDER, AND A. LAYMAN, *Namespaces in XML*, January 1999. W3C Recommendation. <http://www.w3.org/TR/REC-xml-names/>.
- [45] T. BRAY, J. PAOLI, C. M. SPERBERG-MCQUEEN, AND E. MALER, *Extensible Markup Language (XML) 1.0 (second edition)*, October 2000. W3C Recommendation. <http://www.w3.org/TR/REC-xml>.
- [46] R. BROOKS-BILSON, *Programming ColdFusion*, O'Reilly & Associates, August 2001.
- [47] R. E. BRYANT, *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers, 35 (1986), pp. 677–691.
- [48] R. E. BRYANT, *Symbolic boolean manipulation with ordered binary-decision diagrams*, ACM Computing Surveys, 24 (1992), pp. 293–318.
- [49] J. R. BÜCHI, *Weak second-order arithmetic and finite automata*, Zeitschrift für Mathematische Logik und Grundlagen der Mathematik, 6 (1960), pp. 66–92.
- [50] J. R. BÜCHI, *On a decision method in restricted second-order arithmetic*, in Proc. 1st International Congress on Logic, Methodology, and Philosophy of Science, Stanford University Press, 1962.
- [51] J. R. BURCH, E. M. CLARKE, K. L. MCMILLAN, D. L. DILL, AND L. J. HWANG, *Symbolic model checking: 10^{20} states and beyond*, in Proc. 5th Annual IEEE Symposium on Logic in Computer Science, LICS '90, IEEE Computer Society, June 1990.

- [52] W. R. BUSH, J. D. PINCUS, AND D. J. SIELAFF, *A static analyzer for finding dynamic programming errors*, Software: Practice and Experience, 30 (2000), pp. 775–802. John Wiley & Sons.
- [53] K. M. BUTLER, D. E. ROSS, R. KAPUR, AND M. R. MERCER, *Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams*, in Proc. 28th Design Automation Conference, DAC '91, ACM, June 1991.
- [54] P. CAO, J. ZHANG, AND K. BEACH, *Active cache: Caching dynamic contents on the Web*, in Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Middleware '98, Springer-Verlag, September 1998.
- [55] J. CHALLENGER, P. DANTZIG, AND A. IYENGAR, *A scalable system for consistently caching dynamic Web data*, in Proc. 18th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM '99, March 1999.
- [56] D. R. CHASE, M. WEGMAN, AND F. K. ZADECK, *Analysis of pointers and structures*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '90, June 1990.
- [57] A. S. CHRISTENSEN, A. MØLLER, AND M. I. SCHWARTZBACH, *Extending Java for high-level Web service construction*, Tech. Rep. RS-02-11, BRICS, March 2002. Submitted for journal publication. (See Dissertation Chapter 15).
- [58] A. S. CHRISTENSEN, A. MØLLER, AND M. I. SCHWARTZBACH, *Static analysis for dynamic XML*, Tech. Rep. RS-02-24, BRICS, May 2002. Submitted. (See Dissertation Chapter 16).
- [59] J. CLARK, *XSL transformations (XSLT) specification*, November 1999. W3C Recommendation. <http://www.w3.org/TR/xslt>.
- [60] J. CLARK, *TREX – tree regular expressions for XML*, February 2001. <http://www.thaiopensource.com/trex/spec.html>.
- [61] J. CLARK AND S. DEROSE, *XML path language*, November 1999. W3C Recommendation. <http://www.w3.org/TR/xpath>.
- [62] J. CLARK AND M. MURATA, *RELAX NG specification*, December 2001. OASIS. <http://www.oasis-open.org/committees/relax-ng/>.
- [63] C. CONSEL ET AL., *Domain-specific languages*. <http://compose.labri.fr/documentation/dsl/>.
- [64] S. A. COOK AND D. C. OPPEN, *An assertion language for data structures*, in Proc. 2nd ACM Symposium on Principles of Programming Languages, POPL '75, January 1975.

- [65] C. CORBETT, M. B. DWYER, J. HATCLIFF, AND ROBBY, *A language framework for expressing checkable properties of dynamic software*, in Proc. 7th International Workshop on SPIN Software Model Checking, SPIN '00, vol. 1885 of LNCS, Springer-Verlag, August/September 2000.
- [66] T. H. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press, 1990.
- [67] P. COUSOT, *Formal models and semantics*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, MIT Press/Elsevier, 1990, pp. 841–993.
- [68] K. COX, T. BALL, AND J. C. RAMMING, *Lunchbot: A tale of two ways to program Web services*, Tech. Rep. BL0112650-960216-06TM, AT&T Bell Laboratories, 1996.
- [69] N. DAMGAARD, N. KLARLUND, AND M. I. SCHWARTZBACH, *YakYak: Parsing with logical side constraints*, in Developments in Language Theory. Foundations, Applications, and Perspectives, G. Rozenberg and W. Thomas, eds., World Scientific, November 2000, pp. 286–304.
- [70] A. DAVIDSON ET AL., *Schema for object-oriented XML 2.0*, July 1999. W3C Note. <http://www.w3.org/TR/NOTE-SOX/>.
- [71] J. DEAN, D. GROVE, AND C. CHAMBERS, *Optimization of object-oriented programs using static class hierarchy analysis*, in Proc. 9th European Conference on Object-Oriented Programming, ECOOP '95, vol. 952 of LNCS, Springer-Verlag, August 1995.
- [72] S. DEROSE, E. MALER, AND R. DANIEL JR., *XML pointer language*, September 2001. W3C Candidate Recommendation. <http://www.w3.org/TR/xptr>.
- [73] S. DEROSE, E. MALER, AND D. ORCHARD, *XML linking language*, June 2001. W3C Recommendation. <http://www.w3.org/TR/xlink>.
- [74] D. L. DETLEFS, K. R. M. LEINO, G. NELSON, AND J. B. SAXE, *Extended static checking*. Research Report 159, Compaq Systems Research Center, December, 1998.
- [75] A. V. DEURSEN, P. KLINT, AND J. VISSER, *Domain-specific languages*, in The Encyclopedia of Microcomputers, Marcel Dekker, 2002, pp. 53–68.
- [76] J. DONER, *Tree acceptors and some of their applications*, Journal of Computer and System Sciences, 4 (1970), pp. 406–451. Academic Press.
- [77] N. DOR, M. RODEH, AND M. SAGIV, *Checking cleanness in linked lists*, in Proc. 7th International Static Analysis Symposium, SAS '00, vol. 1824 of LNCS, Springer-Verlag, June/July 2000.
- [78] F. DOUGLIS, A. HARO, AND M. RABINOVICH, *HPP: HTML macro-preprocessing to support dynamic document caching*, in Proc. 1st USENIX

- Symposium on Internet Technologies and Systems, USITS '97, December 1997.
- [79] M. DUBINKO, S. SCHNITZENBAUMER, M. WEDEL, AND D. RAGGETT, *XForms requirements*, April 2001. W3C Working Draft. <http://www.w3.org/TR/xhtml1-forms-req.html>.
- [80] J. ELGAARD, *Verifying C pointer programs using monadic second-order logic*, Master's thesis, Department of Computer Science, University of Aarhus, 1999.
- [81] J. ELGAARD, N. KLARLUND, AND A. MØLLER, *MONA 1.x: New techniques for WS1S and WS2S*, in Proc. 10th International Conference on Computer-Aided Verification, CAV '98, vol. 1427 of LNCS, Springer-Verlag, June/July 1998, pp. 516–520. (See Dissertation Chapter 5).
- [82] J. ELGAARD, A. MØLLER, AND M. I. SCHWARTZBACH, *Compile-time debugging of C programs working on trees*, in Programming Languages and Systems, Proc. 9th European Symposium on Programming, ESOP '00, vol. 1782 of LNCS, Springer-Verlag, March/April 2000, pp. 182–194. (See Dissertation Chapter 7).
- [83] C. C. ELGOT, *Decision problems of finite automata design and related arithmetics*, Transactions of the American Mathematical Society, 98 (1961), pp. 21–52.
- [84] D. EVANS, *LCLint user's guide*. <http://lclint.cs.virginia.edu/guide/>.
- [85] D. EVANS, *Static detection of dynamic memory errors*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '96, May 1996.
- [86] M. F. FERNANDEZ, D. SUCIU, AND I. TATARINOV, *Declarative specification of data-intensive Web sites*, in Proc. 2nd Conference on Domain-Specific Languages, DSL '99, USENIX/ACM, October 1999.
- [87] D. FLANAGAN, *JavaScript: The Definitive Guide*, O'Reilly & Associates, June 1998.
- [88] R. W. FLOYD, *Assigning meanings to programs*, Mathematical Aspects of Computer Science, (1967), pp. 19–32. American Mathematical Society.
- [89] J. S. FOSTER, M. FÄHNDRICH, AND A. AIKEN, *A theory of type qualifiers*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '99, May 1999.
- [90] P. FRADET, R. GAUGNE, AND D. LE MÉTAYER, *Static detection of pointer errors: An axiomatisation and a checking algorithm*, in Programming Languages and Systems, Proc. 6th European Symposium on Programming, ESOP '96, vol. 1058 of LNCS, Springer-Verlag, April 1996.

- [91] P. FRADET AND D. LE MÉTAYER, *Shape types*, in Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '97, January 1997.
- [92] C. FRANKSTON AND H. S. THOMPSON, *XML-Data reduced*, July 1998. <http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm>.
- [93] A. O. FREIER, P. KARLTON, AND P. C. KOCHER, *The SSL protocol version 3.0*, November 1996. <http://home.netscape.com/eng/ssl3/draft302.txt>.
- [94] T. GENET AND V. V. T. TONG, *Reachability analysis of term rewriting systems with Timbuk*, Tech. Rep. RR-4266, INRIA Rennes, 2001.
- [95] J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH, AND T. BERNERS-LEE, *Hypertext transfer protocol, HTTP/1.1*, 1999. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>.
- [96] R. GHIYA AND L. J. HENDREN, *Putting pointer analysis to work*, in Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, January 1998.
- [97] A. GIRGENSOHN AND A. LEE, *Seamless integration of interactive forms into the Web*, Computer Networks and ISDN Systems, 29 (1997), pp. 1531–1542. Elsevier. Also in Proc. 6th International World Wide Web Conference, WWW6.
- [98] J. GLENN AND W. I. GASARCH, *Implementing WSIS via finite automata*, in Proc. 1st International Workshop on Implementing Automata, WIA '96, vol. 1260 of LNCS, Springer-Verlag, August 1996.
- [99] GRAMMATECH INC., *CodeSurfer user guide and reference manual*. Available from <http://www.grammatech.com/papers/>.
- [100] D. GRIES, *The Science of Programming*, Springer-Verlag, 1981.
- [101] S. GUNDAVARAM, *CGI Programming on the World Wide Web*, O'Reilly & Associates, March 1996.
- [102] A. GUPTA AND A. L. FISHER, *Representation and symbolic manipulation of linearly inductive boolean functions*, in Proc. International Conference on Computer-Aided Design, ICCAD '93, IEEE Computer Society, November 1993.
- [103] K. HAVELUND AND T. PRESSBURGER, *Model checking Java programs using Java PathFinder*, International Journal on Software Tools for Technology Transfer, 2 (2000), pp. 366–381. Springer-Verlag.
- [104] L. HENDREN, J. HUMMEL, AND A. NICOLAU, *Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '92, June 1992.

- [105] J. G. HENRIKSEN, J. L. JENSEN, M. E. JØRGENSEN, N. KLARLUND, R. PAIGE, T. RAUHE, AND A. SANDHOLM, *Mona: Monadic second-order logic in practice*, in Proc. 1st International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '95, vol. 1019 of LNCS, Springer-Verlag, May 1995.
- [106] C. A. R. HOARE, *An axiomatic basis for computer programming*, Communications of the ACM, 12 (1969), pp. 576–580. ACM.
- [107] A. HOMER, J. SCHENKEN, M. GIBBS, J. D. NARKIEWICZ, J. BELL, M. CLARK, A. ELMHORST, B. LEE, M. MILNER, AND A. REHAN, *ASP.NET Programmer's Reference*, Wrox Press, September 2001.
- [108] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, April 1979.
- [109] P. HOSCHKA ET AL., *Synchronized multimedia integration language (SMIL) 1.0 specification*, June 1998. W3C Recommendation. <http://www.w3.org/TR/REC-smil>.
- [110] H. HOSOYA AND B. C. PIERCE, *XDuce: A typed XML processing language*, in Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB '00, vol. 1997 of LNCS, Springer-Verlag, May 2000.
- [111] H. HOSOYA AND B. C. PIERCE, *Regular expression pattern matching for XML*, in Proc. 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '01, January 2001.
- [112] H. HOSOYA, J. VOUILLON, AND B. C. PIERCE, *Regular expression types for XML*, in Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP '00, September 2000.
- [113] T. HUNE AND A. SANDHOLM, *A case study on using automata in control synthesis*, in Proc. 5th International Conference on Fundamental Approaches to Software Engineering, FASE '00, vol. 1783 of LNCS, Springer-Verlag, March/April 2000.
- [114] J. HUNTER AND B. MCCLAUGHLIN, *JDOM*, 2001. <http://jdom.org/>.
- [115] ICONOCAST INC., *ICONOCAST Newsletter, August 17, 2000*. <http://www.iconocast.com/issue/20000817.html>.
- [116] A. IYENGAR AND J. CHALLENGER, *Improving Web server performance by caching dynamic data*, in Proc. 1st USENIX Symposium on Internet Technologies and Systems, USITS '97, December 1997.
- [117] D. JACKSON, *Aspect: An economical bug-detector*, in Proc. 13th International Conference on Software Engineering, ICSE '91, IEEE Computer Society / ACM, May 1991.

- [118] D. JACKSON AND M. VAZIRI, *Finding bugs with a constraint solver*, in Proc. International Symposium on Software Testing and Analysis, ISSTA '00, ACM, August 2000.
- [119] R. JELLIFFE, *The Schematron: An XML structure validation language using patterns in trees*, 1999. <http://www.ascc.net/xml/resource/schematron/schematron.html>.
- [120] J. L. JENSEN, M. E. JØRGENSEN, N. KLARLUND, AND M. I. SCHWARTZBACH, *Automatic verification of pointer programs using monadic second-order logic*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '97, June 1997.
- [121] B. JONSSON AND M. NILSSON, *Transitive closures of regular relations for verifying infinite-state systems*, in Proc. 6th International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '00, vol. 1785 of LNCS, Springer-Verlag, March/April 2000.
- [122] J. B. KAM AND J. D. ULLMAN, *Monotone data flow analysis frameworks*, Acta Informatica, 7 (1977), pp. 305–317. Springer-Verlag.
- [123] P. KELB, T. MARGARIA, M. MENDLER, AND C. GSOTTBERGER, *MoSeL: A flexible toolset for Monadic Second-order Logic*, in Proc. 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '97, vol. 1217 of LNCS, Springer-Verlag, April 1997.
- [124] F. KLAEDTKE, *Decision procedure for an extension of WSIS*, in Proc. 15th International Workshop on Computer Science Logic, CSL '01, vol. 2142 of LNCS, Springer-Verlag, September 1997.
- [125] N. KLARLUND, *A homomorphism concepts for omega-regularity*, in Proc. 8th International Workshop on Computer Science Logic, CSL '94, vol. 933 of LNCS, Springer-Verlag, September 1997.
- [126] N. KLARLUND, *Mona & Fido: The logic-automaton connection in practice*, in Proc. 11th International Workshop on Computer Science Logic, CSL '97, vol. 1414 of LNCS, Springer-Verlag, August 1997.
- [127] N. KLARLUND, *An $n \log n$ algorithm for online BDD refinement*, Journal of Algorithms, 32 (1999), pp. 133–154. Academic Press. Earlier version in CAV '97, vol. 1254 of LNCS, Springer-Verlag.
- [128] N. KLARLUND, *A theory of restrictions for logics and automata*, in Proc. 11th International Conference on Computer-Aided Verification, CAV '99, vol. 1633 of LNCS, Springer-Verlag, July 1999. An extended version “Relativizations for the Logic-Automaton Connection” has been submitted for publication.
- [129] N. KLARLUND, *From the programmer's point of view: XML for IVR and how DSD schemas may help*. Unpublished revision of “XPML: Industrial Case Study”, currently available at <http://www.research.att.com/projects/DSD/industrial-case/software-symposium-ATT-00-paper/>, September 2000.

- [130] N. KLARLUND, J. KOISTINEN, AND M. I. SCHWARTZBACH, *Formal design constraints*, in Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '96, October 1996.
- [131] N. KLARLUND AND A. MØLLER, *MONA Version 1.4 User Manual*, BRICS, Department of Computer Science, University of Aarhus, January 2001. Notes Series NS-01-1. Available from <http://www.brics.dk/mona/>. Revision of BRICS NS-98-3.
- [132] N. KLARLUND, A. MØLLER, AND M. I. SCHWARTZBACH, *Document Structure Description 1.0*, December 2000. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-00-7. Available from <http://www.brics.dk/DSD/>.
- [133] N. KLARLUND, A. MØLLER, AND M. I. SCHWARTZBACH, *The DSD schema language*, Automated Software Engineering, 9 (2002), pp. 285–319. Kluwer. (See Dissertation Chapter 14). Preliminary version in Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00.
- [134] N. KLARLUND, A. MØLLER, AND M. I. SCHWARTZBACH, *MONA implementation secrets*, International Journal of Foundations of Computer Science, (2002). World Scientific Publishing Company. (See Dissertation Chapter 6). Preliminary version in Proc. 5th International Conference on Implementation and Application of Automata, CIAA '00, vol. 2088 of LNCS, Springer-Verlag.
- [135] N. KLARLUND, M. NIELSEN, AND K. SUNESEN, *Automated logical verification based on trace abstraction*, in Proc. 15th ACM Symposium on Principles of Distributed Computing, PODC '96, May 1996.
- [136] N. KLARLUND, M. NIELSEN, AND K. SUNESEN, *A case study in automated verification based on trace abstractions*, in Formal System Specification, The RPC-Memory Specification Case Study, M. Broy, S. Merz, and K. Spies, eds., vol. 1169 of LNCS, Springer-Verlag, November 1996, pp. 341–374.
- [137] N. KLARLUND AND T. RAUHE, *BDD algorithms and cache misses*, Tech. Rep. RS-96-5, BRICS, 1996.
- [138] N. KLARLUND AND M. I. SCHWARTZBACH, *Graph types*, in Proc. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93, January 1993.
- [139] N. KLARLUND AND M. I. SCHWARTZBACH, *Graphs and decidable transductions based on edge constraints*, in Proc. 19th International Colloquium on Trees in Algebra and Programming, CAAP '94, vol. 787 of LNCS, Springer-Verlag, April 1994.
- [140] N. KLARLUND AND M. I. SCHWARTZBACH, *A domain-specific language for regular sets of strings and trees*, IEEE Transactions on Software Engineering, 25 (1999), pp. 378–386.

- [141] G. KLEIN, *JFlex – the fast scanner generator for Java*, 2001.
<http://www.jflex.de/>.
- [142] A. KOLAWA AND A. HICKEN, *Insure++: A tool to support total quality software*. <http://www.parasoft.com/products/insure/papers/tech.htm>.
- [143] J. KORPELA, *JavaScript and HTML: Possibilities and caveats*, 2000.
<http://www.hut.fi/u/jkorpela/forms/javascript.html>.
- [144] D. A. LADD AND J. C. RAMMING, *Programming the Web: An application-oriented language for hypermedia services*, World Wide Web Journal, 1 (1996). O'Reilly & Associates. Proc. 4th International World Wide Web Conference, WWW4.
- [145] A. LAYMAN ET AL., *XML-Data*, January 1998. W3C Note.
<http://www.w3.org/TR/1998/NOTE-XML-data/>.
- [146] D. LEE AND W. W. CHU, *Comparative analysis of six XML schema languages*, ACM SIGMOD Record, 29 (2000), pp. 76–87.
- [147] T. LEV-AMI, T. REPS, M. SAGIV, AND R. WILHELM, *Putting static analysis to work for verification: A case study*, in Proc. International Symposium on Software Testing and Analysis, ISSTA '00, ACM, August 2000.
- [148] T. LEV-AMI AND M. SAGIV, *TVLA: A system for implementing static analyses*, in Proc. 7th International Static Analysis Symposium, SAS '00, vol. 1824 of LNCS, Springer-Verlag, June/July 2000.
- [149] H. LIEFKE AND D. SUCIU, *XMill: An efficient compressor for XML data*, ACM SIGMOD Record, 29 (2000), pp. 153–164.
- [150] Y. A. LIU, *Efficiency by incrementalization: An introduction*, Higher-Order and Symbolic Computation, 13 (2000), pp. 289–313. Kluwer.
- [151] A. MALHOTRA AND M. MALONEY, *XML Schema requirements*, February 1999. W3C Note. <http://www.w3.org/TR/NOTE-xml-schema-req>.
- [152] T. MARGARIA, *Verification of systolic arrays in M2L(Str)*, Tech. Rep. MIP-9613, Universität Passau, 1996.
- [153] O. MATZ, A. MILLER, A. POTTHOFF, W. THOMAS, AND E. VALKEMA, *Report on the program AMoRE*, Tech. Rep. Report 9507, Inst. für Informatik u. Prakt. Mathematik, CAU Kiel, 1995.
- [154] K. MCMILLAN, *Symbolic Model Checking*, Kluwer, 1993.
- [155] E. MEIJER AND M. SHIELDS, *XM λ : A functional language for constructing and manipulating XML documents*. Draft. Available from <http://www.cse.ogi.edu/~mbs/pub/xmlambda/>, 1999.

- [156] A. R. MEYER, *Weak monadic second-order theory of successor is not elementary recursive*, in Logic Colloquium: Symposium on Logic 1972-73, R. Parikh, ed., vol. 453 of Lecture Notes in Mathematics, Springer-Verlag, 1975, pp. 132–154.
- [157] J. C. MOGUL, F. DOUGLIS, A. FELDMANN, AND B. KRISHNAMURTHY, *Potential benefits of delta encoding and data compression for HTTP*, in Proc. ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '97, September 1997.
- [158] A. MØLLER, *MONA project home page*. <http://www.brics.dk/mona/>.
- [159] A. MØLLER, *PALE project home page*. <http://www.brics.dk/PALE/>.
- [160] A. MØLLER, *Document Structure Description 2.0*. In preparation, 2002.
- [161] A. MØLLER AND M. I. SCHWARTZBACH, *The pointer assertion logic engine*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '01, June 2001. Also in ACM SIGPLAN Notices 36, 5 (May 2001). (See Dissertation Chapter 8).
- [162] A. MØLLER AND M. I. SCHWARTZBACH, *The XML revolution – technologies for the future Web*, December 2001. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-01-8. Available from <http://www.brics.dk/~amoeller/XML/>. Revision of BRICS NS-00-8.
- [163] A. MØLLER AND M. I. SCHWARTZBACH, *Interactive Web services with Java*, April 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-1. Available from <http://www.brics.dk/~amoeller/WWW/>.
- [164] F. MORAWIETZ AND T. CORNELL, *The logic-automaton connection in linguistics*, in Proc. International Conference on Logical Aspects of Computational Linguistics, LACL '97, no. 1582 in LNCS, Springer-Verlag, September 1997.
- [165] F. MORAWIETZ AND T. CORNELL, *On the recognizability of relations over a tree definable in a monadic second order tree description language*, Tech. Rep. SFB 340, Seminar für Sprachwissenschaft Eberhard-Karls-Universität Tübingen, 1997.
- [166] J. M. MORRIS, *A general axiom of assignment*, in Marktoberdorf Summer School on Theoretical Foundations of Programming Methodology, vol. 91 of NATO Science Series, Reidel, August 1982.
- [167] M. MURATA, *Hedge automata: A formal model for XML schemata*, June 1999. http://www.xml.gr.jp/relax/hedge_nice.html.
- [168] M. MURATA, August 2000. Announcement on <http://www.xmlhack.com/>.
- [169] M. MURATA, *How to RELAX*, August 2000. <http://www.xml.gr.jp/relax/>.

- [170] A. MØLLER, *The <bigwig> runtime system*, 2001. <http://www.brics.dk/bigwig/runwig/>.
- [171] A. MØLLER, *dk.brics.automaton – finite-state automata and regular expressions for Java*, 2001. <http://www.brics.dk/automaton/>.
- [172] NETSCAPE CORP., *JavaScript form validation sample code*, 1999. <http://developer.netscape.com/docs/examples/javascript/formval/overview.html>.
- [173] J. NIELSEN, *Designing Web Usability: The Practice of Simplicity*, New Riders Publishing, December 1999.
- [174] F. NIELSON, H. R. NIELSON, AND C. HANKIN, *Principles of Program Analysis*, Springer-Verlag, October 1999.
- [175] M. NILSSON, *Analyzing parameterized distributed algorithms*, Master's thesis, Department of Computer Systems at Uppsala University, Sweden, 1999.
- [176] OPEN MARKET, *FastCGI: A high-performance Web server interface*, April 1996. Available from <http://www.fastengines.com/whitepapers/>.
- [177] G. OSKOBOINY, *HTML Validation Service*, 2001. <http://validator.w3.org/>.
- [178] S. OWRE AND H. RUESS, *Integrating WSIS with PVS*, in Proc. 12th International Conference on Computer-Aided Verification, CAV '00, vol. 1855 of LNCS, Springer-Verlag, July 2000.
- [179] V. PADOVANI, H. COMON, J. LENEUTRE, AND R. TINGAUD, *A formal verification of telephone supplementary service interactions*, Tech. Rep. LSV-99-5, Cachan, France, 1999.
- [180] P. K. PANDYA, *DCVALID 1.4: The user manual*, tech. rep., Tata Institute of Fundamental Research, 2000. Available from <http://www.tcs.tifr.res.in/~pandya/dcvalid.html>.
- [181] P. K. PANDYA, *Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID*, Tech. Rep. TCS00-PKP-1, Tata Institute of Fundamental Research, 2000.
- [182] S. PEMBERTON ET AL., *XHTML 1.0: The extensible hypertext markup language*, January 2000. W3C Recommendation. <http://www.w3.org/TR/xhtml1>.
- [183] M. RABIN, *Decidability of second-order theories and automata on infinite trees*, American Mathematical Society, 141 (1969), pp. 1–35.
- [184] D. RAGGETT, *Assertion grammars*. <http://www.w3.org/People/Raggett/dtdgen/Docs/>, May 1999.
- [185] D. RAGGETT, A. L. HORS, AND I. JACOBS, *HTML 4.01 specification*, December 1999. W3C Recommendation. <http://www.w3.org/TR/html4/>.

- [186] K. RAJAMANI AND A. COX, *A simple and effective caching scheme for dynamic content*, tech. rep., CS Dept., Rice University, September 2000.
- [187] A. S. RASMUSSEN, *Symbolic model checking using monadic second order logic as requirement language*, Master's thesis, Technical University of Denmark (DTU), 1999. IT-E 821.
- [188] RATIONAL SOFTWARE CORP., *Purify*. <http://www.rational.com/>.
- [189] J. C. REYNOLDS, *Intuitionistic reasoning about shared mutable data structure*, in *Millennial Perspectives in Computer Science*, Proc. 1999 Oxford–Microsoft Symposium in Honour of Sir Tony Hoare, J. Davies, B. Roscoe, and J. Woodcock, eds., Palgrave, November 2000, pp. 303–321.
- [190] M. RICKY, *Automatisk validering af webbaserede formularer*, Master's thesis, Department of Computer Science, University of Aarhus, 2002. (In Danish).
- [191] J. ROBIE, *W3C XML Schema questionnaire*, July 2000. <http://www.ibiblio.org/xql/tally.html>.
- [192] M. SAGIV, T. REPS, AND R. WILHELM, *Solving shape-analysis problems in languages with destructive updating*, *ACM Transactions on Programming Languages and Systems*, 20 (1998), pp. 1–50. ACM.
- [193] M. SAGIV, T. REPS, AND R. WILHELM, *Parametric shape analysis via 3 valued logic*, in Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '99, January 1999.
- [194] A. SANDHOLM AND M. I. SCHWARTZBACH, *Distributed safety controllers for Web services*, in Proc. 3rd International Conference on Fundamental Approaches to Software Engineering, FASE '98, vol. 1382 of LNCS, Springer-Verlag, March/April 1998.
- [195] A. SANDHOLM AND M. I. SCHWARTZBACH, *A type system for dynamic Web documents*, in Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00, January 2000.
- [196] S. SCHNITZENBAUMER, M. WEDEL, AND M. GUNATILAKE, *XHTML-FML 1.0: Forms markup language*, 1999. Stack Overflow AG. http://www.mozquito.org/documentation/spec_xhtml-fml.html.
- [197] M. I. SCHWARTZBACH ET AL., *<bigwig> project home page*. <http://www.brics.dk/bigwig/>.
- [198] T. R. SHIPLE, J. H. KUKULA, AND R. K. RANJAN, *A comparison of Presburger engines for EFSM reachability*, in Proc. 10th International Conference on Computer-Aided Verification, CAV '98, vol. 1427 of LNCS, Springer-Verlag, June/July 1998.
- [199] B. SMITH, A. ACHARYA, T. YANG, AND H. ZHU, *Exploiting result equivalence in caching dynamic Web content*, in Proc. 2nd USENIX Symposium on Internet Technologies and Systems, October 1999.

- [200] M. A. SMITH AND N. KLARLUND, *Verification of a sliding window protocol using IOA and Mona*, in Proc. Formal Techniques for Distributed System Development, FORTE '00, vol. 183 of IFIP Conference Proceedings, Kluwer, October 2000.
- [201] SUN MICROSYSTEMS, *Java Servlet Specification, Version 2.3*, 2001. Available from <http://java.sun.com/products/servlet/>.
- [202] SUN MICROSYSTEMS, *JavaServer Pages Specification, Version 1.2*, 2001. Available from <http://java.sun.com/products/jsp/>.
- [203] V. SUNDARESAN, L. J. HENDREN, C. RAZAFIMAHEFA, R. VALLEE-RAI, P. LAM, E. GAGNON, AND C. GODIN, *Practical virtual method call resolution for Java*, in Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00, October 2000.
- [204] S. TANI, K. HAMAGUCHI, AND S. YAJIMA, *The complexity of the optimal variable ordering problems of a shared binary decision diagram*, in Proc. International Conference on Computer-Aided Design, ICCAD '93, IEEE Computer Society, November 1993.
- [205] J. W. THATCHER AND J. B. WRIGHT, *Generalized finite automata with an application to a decision problem of second-order logic*, Mathematical Systems Theory, 2 (1968), pp. 57–82. Springer-Verlag.
- [206] THE UNICODE CONSORTIUM, *The Unicode Standard, Version 2.0*, Addison Wesley, 1996. <http://www.unicode.org/>.
- [207] P. THISTLEWAITE AND S. BALL, *Active FORMs*, Computer Networks and ISDN Systems, 28 (1996), pp. 1355–1364. Elsevier. Also in Proc. 5th International World Wide Web Conference, WWW5.
- [208] W. THOMAS, *Automata on infinite objects*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, MIT Press/Elsevier, 1990, pp. 133–191.
- [209] H. S. THOMPSON, D. BEECH, M. MALONEY, AND N. MENDELSON, *XML Schema part 1: Structures*, May 2001. W3C Recommendation. <http://www.w3.org/TR/xmlschema-1/>.
- [210] F. TIP, *A survey of program slicing techniques*, Journal of Programming Languages, 3 (1995), pp. 121–189. CompSciNet.
- [211] B. A. TRAKHTENBROT, *Finite automata and the logic of one-place predicates*, Siberian Mathematical Journal, 3 (1962), pp. 103–131. English translation in AMS Transl. 59 (1966), 23–55.
- [212] M. TSIMELZON, B. WEIHL, AND L. JACOBS, *ESI language specification 1.0*. <http://www.edge-delivery.org/language-spec.1-0.html>, 2001.

- [213] R. VALLEE-RAI, L. HENDREN, V. SUNDARESAN, P. LAM, E. GAGNON, AND P. CO, *Soot – a Java optimization framework*, in Proc. IBM Centre for Advanced Studies Conference, CASCON '99, IBM, November 1999.
- [214] A. VAN DEURSEN, P. KLINT, AND J. VISSER, *Domain-specific languages: An annotated bibliography*, ACM SIGPLAN Notices, 35 (2000), pp. 26–36.
- [215] J. WALDMANN, *Tree automata in RX*. Software demo at 3rd International Workshop on Implementing Automata, WIA '99, 1999.
- [216] J. WANG, *A survey of Web caching schemes for the Internet*, ACM Computer Communication Review, 29 (1999), pp. 36–46.
- [217] WAP FORUM, *Wireless Markup Language, version 2.0*, September 2001. Wireless Application Protocol Forum. Available from <http://www.wapforum.org/>.
- [218] M. WEBB AND M. PLUNGJAN, *JavaScript form FAQ knowledge base*, 2000. <http://developer.irt.org/script/form.htm>.
- [219] D. WEISE AND R. F. CREW, *Programmable syntax macros*, in Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '93, June 1993.
- [220] C. WILLS AND M. MIKHAILOV, *Studying the impact of more complete server information on Web caching*, Computer Communications, 24 (2001), pp. 184–190. Elsevier. Also in Proc. 5th International Web Caching and Content Delivery Workshop.
- [221] A. WOLMAN, *Characterizing Web workloads to improve performance*, July 2000. University of Washington. Available from <http://www.cs.washington.edu/homes/wolman/generals/>.
- [222] K. YAGOUB, D. FLORESCU, V. ISSARNY, AND P. VALDURIEZ, *Caching strategies for data-intensive Web sites*, in Proc. 26th International Conference on Very Large Data Bases, VLDB '2000, Morgan Kaufmann, September 2000.
- [223] S. YANG, *Logic synthesis and optimization benchmarks user guide, version 3.0*, Tech. Rep. 1991-IWLS-UG-Saeyang, Microelectronics Center of North Carolina, 1991.
- [224] H. ZHU AND T. YANG, *Class-based cache management for dynamic Web contents*, in Proc. 20th Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM '01, April 2001, pp. 1215–1224.

Recent BRICS Dissertation Series Publications

- DS-02-4 Anders Møller. *Program Verification with Monadic Second-Order Logic & Languages for Web Service Development*. September 2002. PhD thesis. xvi+337 pp.
- DS-02-3 Riko Jacob. *Dynamic Planar Convex hull*. May 2002. PhD thesis. xiv+112 pp.
- DS-02-2 Stefan Dantchev. *On Resolution Complexity of Matching Principles*. May 2002. PhD thesis. xii+70 pp.
- DS-02-1 M. Oliver Möller. *Structure and Hierarchy in Real-Time Systems*. April 2002. PhD thesis. xvi+228 pp.
- DS-01-10 Mikkel T. Jensen. *Robust and Flexible Scheduling with Evolutionary Computation*. November 2001. PhD thesis. xii+299 pp.
- DS-01-9 Flemming Friche Rodler. *Compression with Fast Random Access*. November 2001. PhD thesis. xiv+124 pp.
- DS-01-8 Niels Damgaard. *Using Theory to Make Better Tools*. October 2001. PhD thesis.
- DS-01-7 Lasse R. Nielsen. *A Study of Defunctionalization and Continuation-Passing Style*. August 2001. PhD thesis. iv+280 pp.
- DS-01-6 Bernd Grobauer. *Topics in Semantics-based Program Manipulation*. August 2001. PhD thesis. ii+x+186 pp.
- DS-01-5 Daniel Damian. *On Static and Dynamic Control-Flow Information in Program Analysis and Transformation*. August 2001. PhD thesis. xii+111 pp.
- DS-01-4 Morten Rhiger. *Higher-Order Program Generation*. August 2001. PhD thesis. xiv+144 pp.
- DS-01-3 Thomas S. Hune. *Analyzing Real-Time Systems: Theory and Tools*. March 2001. PhD thesis. xii+265 pp.
- DS-01-2 Jakob Pagter. *Time-Space Trade-Offs*. March 2001. PhD thesis. xii+83 pp.
- DS-01-1 Stefan Dziembowski. *Multiparty Computations — Information-Theoretically Secure Against an Adaptive Adversary*. January 2001. PhD thesis. 109 pp.