BRICS RS-04-34 Møller & Schwartzbach: The Design Space of Type Checkers for XML Transformation Languages

BRICS

Basic Research in Computer Science

The Design Space of Type Checkers for XML Transformation Languages

Anders Møller
Michael I. Schwartzbach

BRICS Report Series RS-04-34
ISSN 0909-0878 December 2004

See back inner page for a list of recent BRICS Report 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 `RS/04/34/`

The Design Space of Type Checkers for XML Transformation Languages

Anders Møller⋆ and Michael I. Schwartzbach

BRICS⋆⋆, University of Aarhus
{amoeller,mis}@brics.dk

Abstract. We survey work on statically type checking XML transformations, covering a wide range of notations and ambitions. The concept of *type* may vary from idealizations of DTD to full-blown XML Schema or even more expressive formalisms. The notion of *transformation* may vary from clean and simple transductions to domain-specific languages or integration of XML in general-purpose programming languages. Type annotations can be either explicit or implicit, and type checking ranges from exact decidability to pragmatic approximations.
We characterize and evaluate existing tools in this design space, including a recent result of the authors providing practical type checking of full unannotated XSLT 1.0 stylesheets given general DTDs that describe the input and output languages.

1 Introduction

XML is an established format for structured data, where each document is essentially an ordered labeled tree [8]. An XML *language* is a subset of such trees, typically described by formalisms known collectively as *schemas*. Given a schema S, we use $\mathcal{L}(S)$ to denote the set of XML trees that it describes. Several different schema formalisms have been proposed: the original DTD mechanism that is part of the XML specification [8], more expressive schemas such as XML Schema [39], RELAX NG [13], or DSD2 [33], and various tree automata formalisms [21, 14].

Many different languages have been devised for specifying transformations of XML data, covering a wide range of programming paradigms. Several such languages have type systems that aim to statically catch runtime errors that may occur during transformations, but not all consider the overall problem of type checking the global effect: given a transformation T, an input schema S_{in} and an output schema S_{out}, decide at compile time if

$$\forall X \in \mathcal{L}(S_{in}) : T(X) \in \mathcal{L}(S_{out})$$

The input and output language may of course be the same. Notice that schemas here act as types in the programming language. Also, we use the notion of *type checking* in a general sense that also covers techniques based on dataflow analysis.

⋆ Supported by the Carlsberg Foundation contract number ANS-1507/20.
⋆⋆ Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

In this paper we describe the design space of XML transformation languages and their type checkers, and survey a representative collection of examples: XDuce [21], XACT [27], XJ [17], XOBE [24], JDOM [23], JAXB [37], HaXml [42], Cω [31, 4], *tree transducers* [32, 30], and XQuery [5]. Furthermore, we present a preliminary report on a novel point in the design space: a flow-based type checker for the full XSLT 1.0 language [34].

XML transformations are motivated by different usage scenarios: queries on XML databases generate results that are again XML data; XML documents are presented in XHTML or XSL-FO versions; translations are performed between different dialects of XML languages; and views or summaries of XML publications are automatically extracted. A major contributing factor to the success of XML is the ability to unify such diverse tasks in a single framework. While the various languages we survey certainly have different sweet spots, it is reasonable to expect that they should each support most of the above scenarios.

2 Aspects of XML Transformation Languages

Languages for programming XML transformations may be characterized in many different ways. A major distinction, which is actually relevant for any kind of application domain, is between *domain-specific languages (DSLs)* and *general purpose languages (GPLs)* [41].

A **GPL** is an ordinary programming language, like Java or C++. One approach for obtaining integration of XML is using a library that allows construction and deconstruction of XML values and provides support for parsing and unparsing. Another approach is *data binding*, which represents XML data through native types of the programming language, often guided by schemas and manually specified mappings. Since XML documents are ordinary data values in the language, there is no special syntax or analysis of XML manipulations.

A DSL is a specially designed language that supports domain-specific values and operations. A DSL may be either *stand-alone* or *embedded*.

A **stand-alone DSL** is designed and implemented from scratch with its own tailor-made compiler or interpreter. This allows the highest degree of exploitation of domain-specific knowledge for both syntax, analysis, and implementation. Stand-alone DSLs have two obvious downsides: First, it is expensive to implement a language from scratch (though there is much active research in lowering the cost) and it is difficult to provide as complete an infrastructure as, say, the Java language libraries. Second, potential users have a steep learning curve, even though DSLs are often designed to resemble other known languages. For XML transformations, a stand-alone DSL will have some kind of XML trees as native values.

An **embedded DSL** is based on a GPL with a framework for XML programming. The domain-specific parts consist of specialized syntax and analysis. The domain-specific syntax, typically for XML constants and navigation in XML data, may be provided through a preprocessor that desugars the DSL syntax into GPL syntax. At runtime, the DSL operations are then handled by a GPL library.

The domain-specific program analysis may be performed at the DSL source code level, on the desugared code, or it may exploit a GPL analysis. Compared to a stand-alone DSL, having the foundation of a GPL makes it easier to interact with other systems, for example, communicate on the Web or access data bases through the use of preexisting libraries. It also makes it simpler to integrate non-XML computations into the XML processing, for example, complex string manipulations or construction of XML data from non-XML data or vice versa.

The distinction between these categories can be blurry: a GPL may be extended with XML-like features to allow better data binding, or a DSL for XML processing may be extended to become a GPL. We shall call these approaches **XML-centric GPLs**.

Another distinguishing aspect of an XML transformation language is its **expressiveness**. All GPLs are clearly Turing complete, but some embedded DSLs are designed to express only a limited class of transformations. The benefit of a restricted language is twofold: First, if only certain transformations are required, then a specialized syntax makes their programming easier. Second, a restricted language may be subjected to more precise analysis.

It is important to distinguish between two different kinds of Turing completeness: the ability to perform arbitrary computations on representations of XML trees vs. the ability to perform arbitrary computations on some encoding of the integers. Some stand-alone DSLs are only Turing complete in the latter sense. Also, an embedded DSL may be Turing incomplete in the XML sense, even though the underlying GPL is Turing complete in the traditional sense. A common example is a language where element names and attribute names cannot be generally computed but only chosen among constants that appear in the program. However, such a restriction might not be a limitation in practice since schemas written in existing schema languages can only define fixed sets of such names anyway.

A well-known aspect of any language is its **paradigm**. For the purpose of this paper, we shall merely distinguish roughly between *imperative* languages, which have explicit mutable state, and *declarative* languages, which are without side-effects and often have implicit control flow. The most widely used GPLs are imperative, but most stand-alone DSLs are declarative.

The languages we consider apply an abundance of different **models of XML data**. Some treat XML as *mutable* tree structures, whereas others view them as being *immutable*. Of course, this aspect is closely related to the language paradigm. Mutability is natural in an imperative language; however, immutability has many advantages, in particular with respect to type checking, and can be beneficial even in languages where data is generally mutable. In approaches that involve data binding for GPLs, XML data is represented using the underlying data model, in object-oriented languages by mapping schema types into classes and XML data into objects.

These different models may involve varying mechanisms for constructing XML data and for navigating through or deconstructing the data. One approach is to perform direct tree manipulation where construction and navigation is on the level of individual XML tree nodes, perhaps with a term language for constructing tree fragments from constants and dynamically evaluated expressions. Deconstruction and navigation might also be based on *pattern matching* or on XPath expressions. Another variant is to model XML data as *templates*, which are tree structures with gaps that can be substituted with other values.

Finally, the **quality of the implementation** of a language may obviously vary. We will distinguish between three levels: First, an *industrial strength* implementation scales to real-life applications and has a robust infrastructure of support and documentation. Second, a *prototype* is provided by a research group, has been shown to work on moderately sized examples, and is only sporadically supported. Finally, *theoryware* is an implementation whose feasibility has been established in a research paper but for which little practical experience exists. Note that many prominent software tools have taken the full trip from theoryware over prototype to industrial strength.

3 Aspects of XML Type Checking

Independently of the above aspects, the type checking capabilities of an XML transformation language may be characterized.

First of all, an XML transformation language may of course be entirely unchecked, which means that all errors will be observed during runtime. For the other languages, we will distinguish between *internal* and *external* type checking.

Internal type checks aim at eliminating runtime errors during execution. For a GPL framework, this property is mainly inherited from the underlying language, and XML operations throwing their own kinds of exceptions are generally beyond the scope of the type system. For an embedded DSL, the type system of the underlying language will perform its own checks, while a separate analysis may perform additional checks of XML operations. One example of this is to verify that when the program navigates to, say, a given attribute of an element, then such an attribute is guaranteed to exist according to the schema. For a stand-alone DSL, a domain-specific type checker is often integrated into the compiler.

External type checks aim at validating the overall behavior of the XML transformation: that an XML tree belonging to the language of an input schema is always transformed into an XML tree belonging to the language of an output schema. For a GPL framework, this will require a global program analysis of the underlying language. This is often also true for an embedded DSL, but the restrictions imposed by the domain-specific syntax may make this task considerably simpler. For a stand-alone DSL, the external type check may also require a program analysis, but often it will be possible to express external type checks in terms of internal type checks if the schemas can be mapped to the domain-specific types.

The types of the XML data must be specified in some **type formalism**. A simple choice is DTD or the closely related formalism of *local tree grammars* [35]. A more ambitious choice is to use general *regular (unranked) tree languages*, corresponding to bottom-up tree automata [36]. Another approach is to use the full XML Schema language or other advanced schema languages used in real-life development projects. Finally, for GPL frameworks and embedded DSLs, the types will be characterized as *native* if they are effectively those of the underlying language. Approaches that rely on schema languages such as DTD or XML Schema most often tacitly ignore uniqueness and reference constraints (that is, ID/IDREF in DTD and key/keyref/unique in XML Schema), since these aspects of validity seem exceedingly hard to capture by type systems or dataflow analysis, and also usually are regarded as secondary features compared to the structural aspects of schemas.

Some type checkers use **type annotations**, which are part of the language syntax and explicitly state the expected or required types of variables and expressions. Annotations may be mandatory, meaning that certain language constructs must be given explicit types. Some languages require annotation of every XML variable, whereas others have a more light use of annotations, for example at input and output only. Heavy use of type annotations has both pros and cons. Type annotations may make the task of the type checker easier since less inference is needed. Also, one may argue that explicit types make the code more comprehensible since its intention is made more clear. On the other hand, the types being involved in XML processing can be quite complicated and writing explicit types might be viewed as an annoying extra burden on the programmer. Also, explicit types may incur a *rigid* type checker where type correctness must be obeyed at every program point. The consequences might be that XML trees can only be built strictly bottom-up, and that sequences of updates that gradually convert data from one type to another are not possible to type check. We discuss these issues further in the next section.

For Turing complete languages, type checking is an undecidable problem. For internal type checks, the decision problem is to determine the absence of certain runtime errors. For external type checks, the decision problem is to determine if the input language is transformed into the output language. Thus, type systems must approximate the answers to these problems. We will characterize the **precision** of both the internal and the external type checking capabilities according to the levels of guarantees being provided: The typical solution is to devise a static type checking algorithm that *conservatively* (that is, soundly but not completely) decides if the desired properties hold. Thus, any type checker will unfairly reject some programs, which is a common experience of most programmers. Another solution is to apply a *pragmatic* type checker which attempts to catch as many errors as possible, but which may generate both false positives and false negatives (in other words, it is neither sound nor complete). Note that even conservative internal type checkers usually ignore certain kinds of runtime errors, the classical examples being division by zero and null pointer dereferences. Also, approaches belonging to the pragmatic category can be sound if certain unchecked assumptions are adhered to. Of course, for non-Turing complete languages, it might also be feasible to guarantee *exact* answers.

The theoretical **complexity** of the type checking algorithm is also a relevant aspect. However, the asymptotic complexity of an algorithm is not always a true measure of its experienced running time (for example, ML type inference is exponential but runs smoothly in practice). A related aspect is the **modularity** of the type checking. A highly modular approach is more likely to scale to large programs. Some algorithms analyze each operation individually; if each operation type checks, then the entire program type checks. Others involve whole-program type inference or dataflow analysis using fixed-point iteration. Naturally, this aspect depends on the use of type annotations described above: high modularity is correlated with heavy use of annotations.

Finally, as for the transformation implementation, we will characterize the **type checking implementation**; for some languages, the transformation implementation is much more developed than the type checker. The **availability** of implementations is also interesting, where we distinguish between *open source*, free *binary* distributions, *commercial* products, and implementations that seem *unavailable*.

4 Points in the Design Space

The above discussions allow us to provide a succinct profile of a given XML transformation language and its type checker. For each language, we look into the following aspects (however, some are not applicable to all examples):

Language type: Is the language a GPL library, a data-binding framework, a stand-alone DSL, an embedded DSL, or an XML-centric GPL? In case of a stand-alone DSL, is it imperative or declarative? Is it Turing complete?

Model for XML data: Is XML data mutable or immutable? How is XML data constructed and deconstructed?

Type formalism: Which formalism is used as types?

Annotations: How much does the approach depend on explicit types in the programs?

Precision: Is the type system exact, conservative, or pragmatic? Which guarantees are given when a program type checks? This aspect is relevant for both internal and external type checks. For conservative systems, is the precision acceptable in practice or are too many semantically correct programs rejected by the type checker?

Complexity: What is the theoretical complexity of the type checking process (if known)? Of course, this aspect must be evaluated together with the modularity aspect. Also, observed behavior in practice may appear very different.

Modularity: What is the granularity of the type checking? This ranges from individual operations to whole-program analyses.

Implementation quality and availability: What is the quality of implementations of the transformation language and of the type checker? Is their source code available?

Additionally, we will try to relate each language with the most closely related ones to investigate the similarities and essential differences.

4.1 XDuce

XDuce was the first programming language with type checking of XML operations using schemas as types [21]. It is a simplistic language that has provided the foundation for later languages, in particular Xtatic and ℂDuce, which we briefly mention below, and has also influenced the design of XQuery (see Section 4.10) and the popular schema language RELAX NG [13].

Language type: XDuce is a declarative stand-alone DSL. It can also be characterized as a first-order pure functional language. Its intention has been to investigate type-safe integration of XML into programming languages, not to be a full fledged programming language. The original description of the language did not include attributes, but this has been amended in a later version [19]. It is Turing complete, with the exception that it cannot compute element names and attribute names dynamically.

Model for XML data: Since the language is pure, XML data is obviously treated as immutable trees. Construction of values is expressed as tree terms. Navigation and deconstruction is based on a mechanism of *regular expression pattern matching* [20] – a combination of regular expressions and ML-style pattern matching that is closely connected with the type system.

Type formalism: The type system of XDuce is based on the notion of *regular expression types*, which corresponds to the class of regular tree languages. The most essential part of the type system is the subtyping relation, which is defined by inclusion of the values represented by the types (this is also called *structural* subtyping).

Annotations: XDuce requires explicit type annotations for both function arguments and return values; however it provides local type inference for pattern matching operations, which means that many pattern variables do not need annotations.

Precision: The type checker of XDuce is conservative: a program that passes type checking is guaranteed to transform valid input into valid output. Regarding internal checking, various properties of pattern matching operations are checked: exhaustiveness (that at least one clause always matches), irredundancy (every clause can match some value), and unambiguity (that unique bindings are always obtained). Since the type formalism is decidable there exist programs that are semantically correct but where appropriate type annotations are not expressible, but such problematic programs have not been described in the XDuce papers.

Complexity: Since subtyping is based on automata language inclusion, the complexity of type checking—including the local type inference and the checks of pattern matching operations—is exponential time complete. Nevertheless, the algorithm being used appears efficient in practice [22].

Modularity: Since no global type inference or fixed-point iteration is involved, the approach is highly modular.

Implementation quality and availability: An open source prototype is available. This implementation focuses on type checking and analysis of patterns, not on runtime efficiency.

A key to the success of XDuce is the clean mathematical foundation of regular expression types. However, a number of issues remain. First, the current design does not handle unordered content models although these are common in real-life schemas. Second, the regular expression pattern matching mechanism can in some situations be too low-level, for example, for navigating deep down XML tree structures, processing data iteratively, or performing almost-identity transformations. Ongoing work aims to provide higher-level pattern matching primitives [18]. Third, devising an efficient runtime model for the language is challenging; for example, pattern matching may involve exponential time or space algorithms [29].

Other issues are being addressed in descendants of XDuce: Xtatic [15] aims to integrate the main ideas from XDuce into C$^\sharp$ (and can hence be categorized as an embedded DSL). As a part of making the technologies available in a mainstream language, efficient runtime representation of the XML data is also considered [16]. The $\mathbb{C}$Duce language [3] goes another direction by extending XDuce towards being an XML-centric functional GPL by adding features, such as higher-order functions and variations of pattern matching primitives. Additionally, parametric polymorphism is being considered.

4.2 Xact

Xact [27, 26] has roots in the language JWIG, which is a Java-based language for development of interactive Web services [10, 7]. JWIG contains a template-based mechanism for dynamic construction of HTML/XHTML pages and includes a static program analysis that checks for validity of the pages; in Xact this mechanism has been generalized to full XML transformations.

Language type: Xact is an embedded DSL, with Java as host language. As XDuce, it is Turing complete but cannot compute element names and attribute names dynamically.

Model for XML data: This language uses a variant of immutable trees called *templates*, which are XML tree fragments with named *gaps* appearing in element contents or attributes. Values can be filled into these gaps in any order and at any time, and conversely, subtrees can be replaced by gaps in order to remove or replace data. Constant templates are written in an XML syntax. The main operations are the following: `plug` constructs a new value by inserting XML templates or strings into the gaps of the given name; `select` takes an XPath expression as argument and returns an array of the selected subtrees; `gapify` also takes an XPath expression as argument but in contrast to `select` it replaces the addressed subtrees by gaps of a given name; `setAttribute` inserts or replaces attributes selected using XPath; and `setContent` similarly replaces element content. In addition, there are methods for importing and exporting XML values to other formats, such as strings, streams, or JDOM documents. Note that a major difference to the XDuce family of languages is that Xact relies on XPath for navigation in XML trees.

Type formalism: The static guarantees in XACT are obtained through the use of a dataflow analysis that exploits a formalism called *summary graphs*, which approximatively tracks the operations on templates in the program. DTD is used for input and output types; however, the analyzer does permit the stronger schema language DSD2 [33] for the output types. The asymmetry arises since the input type must be translated into a summary graph, while the final check of the output type uses a separate algorithm that tests inclusion of summary graphs into DSD2 schemas. It is theoretically possible to map also a DSD2 schema into a summary graph accepting the same language (ignoring as usual uniqueness and pointer constraints), but this has not been implemented yet.

Annotations: Being based on dataflow analysis, the annotation overhead is much lighter than in most other techniques. Types, that is, references to DTDs, are specified only at input and at designated analysis points (typically at output).

Precision: The analysis is conservative, that is, a program that passes the analysis cannot produce invalid XML at runtime. The analyzer also performs some internal checks: that `plug` operations never fail (by attempting to plug templates into attribute gaps), and that XPath expressions used in the other XML operations can potentially select nonempty node sets. The main practical limitations of the analysis precision are caused by the facts that the current implementation employs a monovariant and path-insensitive analysis and that all field variables are treated flow insensitively (to ensure soundness).

Complexity: The analysis has polynomial complexity.

Modularity: The approach has poor modularity since it performs fixed-point iteration over the entire program. Nevertheless, it appears reasonably efficient in practice [27].

Implementation quality and availability: An open source prototype is available. The analyzer handles the full Java language. The runtime representation has been crafted to obtain good performance despite operating on immutable structures. [26]

Although less mathematically elegant, the template-based mechanism in XACT can be more flexible to program with than the XDuce model. First, using the `plug` operation, templates with gaps can be passed around as first-class values. Gaps may be filled in any order and computed templates can be reused; in the XDuce family of languages, trees must be constructed bottom-up. Second, the use of XPath appears powerful for addressing deeply into XML trees; several other languages have chosen XPath for the same purpose, as described in the following sections. Third, the `gapify` operation makes it easy to make almost-identity transformations without explicitly reconstructing everything that does not change.

Despite the differences between the XDuce and XACT approaches, there is a connection between the underlying formalisms used in the type checkers: as shown in [9], the notions of summary graphs and regular expression types are closely related.

Current work on the XACT project aims to obtain a closer integration with the new generics and iteration features that have been introduced in Java 5.0.

4.3 XJ

The development of the XJ [17] language aims at integrating XML processing closely into Java using XML Schema as type formalism.

Language type: XJ is an embedded DSL using Java as host language.

Model for XML data: XML data is represented as mutable trees. Construction of XML data is performed at the level of individual nodes. It is dynamically checked that every node has at most one parent. Subtrees are addressed using XPath. Updating attribute values or character data is likewise expressed using XPath, whereas insertion and deletion of subtrees involving elements are expressed with special `insert` and `delete` operations.

Type formalism: Types are regular expressions over XML Schema declarations of elements, attributes, and simple types. Thus, the type system has two levels: regular expression operators and XML Schema constructions. Subtyping on the schema level is defined by the use of type derivations (extensions and restrictions) and substitution groups in the schemas: if A is derived from B or is in the substitution group of B, then A is defined to be a subtype of B. In other words, this is a *nominal* style of subtyping. Subtyping on the regular expression level is defined as regular language inclusion on top of the schema subtyping. Coercions are made between certain XML types and normal Java types, for example between `int` of XML Schema and `int` of Java, or between Kleene star and `java.lang.List`.

Annotations: All XML variable declarations must be annotated with types.

Precision: The type checker is in the pragmatic category because updates require runtime checks due to potential aliasing. Also, not all features in XML Schema are accounted for by the type system, an example being facet constraints. Updates involving XPath expressions that evaluate to multiple nodes result in runtime errors.

Complexity: Since subtyping relies on inclusion between regular expressions, complexity is exponential in the size of the regular expressions being used.

Modularity: Due to the heavy use of annotations, each operation can be checked separately, which leads to a high degree of modularity.

Implementation quality and availability: A prototype implementing parts of the system has been made (in particular, type checking of updates of complex types is not implemented). This prototype has not been available to us.

The authors of [17] acknowledge the fact that the type checker of XJ can be too rigid. Since the values of a variable at all times must adhere to its type and this type is fixed at the variable declaration, it is impossible to type check a sequence of operations that temporarily invalidate the data. A plausible example is constructing an element and inserting a number of mandatory attributes through a sequence of updates.

A problem with the nominal style of subtyping is that a given XML value is tied too closely with its schema type. Imagine a transformation (inspired by the `addrbook` example from [21]), which creates a telephone book document from an address book document by extracting the entries that have telephone numbers. That is, the output language is a subset of the input language, the only difference being that `telephone` elements are mandatory in the content model of `person` elements in the output. Since the nominal type system treats the two versions of `person` elements as unrelated, an XJ transformation must explicitly reconstruct all `person` elements instead of merely removing those without a `telephone` element.

4.4 XOBE

The XOBE language [24] has been developed with similar goals as XJ and has many similarities in the language design; however, the type checking approach appears closer to that of XDuce.

Language type: XOBE is an embedded DSL using Java as host language.

Model for XML data: XML data is represented as mutable trees. (It is not explicitly stated in the available papers on XOBE that the XML trees are mutable, however an example program in [24] strongly suggests that this is the case.) Construction of XML trees is written in an XML-like notation with embedded expressions (unlike XJ). Subtrees are addressed using XPath expressions.

Type formalism: The underlying type formalism is *regular hedge expressions*, which corresponds to the class of regular tree languages that, for instance, XDuce also relies on. From the programmer's point of view, XML Schema can be used as type formalism, but features of XML Schema that go beyond regularity are naturally not handled by the type checker. It is not clear how the type derivation and substitution group features of XML Schema are handled, but it might be along the lines suggested in [35].

Annotations: XOBE requires explicit type annotations on every XML variable declaration.

Precision: The main ingredient of the type checker is checking subtype relationship for assignment statements. Since mutable updates are possible and the potential aliases that then may arise are apparently ignored (unlike in the XJ approach which relies on runtime checks), the XOBE type checker is unsound and hence belongs in the pragmatic category. However, if assuming that problematic aliases do not arise, type checking is conservative. When XML Schema is used as type formalism, certain kinds of constraints that are expressible in XML Schema, such as number of occurrences and restricted string types, are handled by runtime checks.

Complexity: The complexity of checking subtype relationship is exponential.

Modularity: As with XJ, the modularity of type checking is high.

Implementation quality and availability: A binary-code prototype is available (but, at the time of writing, with minimal documentation).

As in XJ, integrating XML into a GPL using mutable trees as data model and XPath for addressing subtrees is a tempting and elegant approach. However, two crucial problems remain: it appears infeasible to ensure soundness of the type checker when aliasing and updates can be mixed, and the type checker can be too rigid as noted above.

4.5 JDOM

As as baseline, we include JDOM [23] – a popular approach that does not perform any type checking for validity of the generated XML data but only ensures well-formedness. In return, it is simple, offers maximal flexibility and performance, and is widely used. JDOM is developed as a Java-specific alternative to the language independent DOM [1].

Language type: JDOM is a GPL library (for Java).

Model for XML data: XML data is represented as mutable trees (in particular, nodes must have unique parents). The library contains a plethora of operations for performing low-level tree navigation and manipulation and for importing and exporting to other formats. Additionally, there is built-in support for evaluating XPath location path expressions.

Type formalism: Well-formedness comes for free with the tree representation, but JDOM contains no type system (in addition to what Java already has).

Implementation quality and availability: JDOM has an open source industrial strength implementation.

Compared to the other approaches mentioned in this paper, DOM and JDOM are generally regarded as low-level frameworks. They are often used as foundations for implementing more advanced approaches.

4.6 JAXB

Relative to DOM/JDOM, Sun's JAXB framework [37] and numerous related projects [6] can be viewed as the next step in integrating XML into GPL programming languages.

Language type: JAXB is a data binding framework for Java.

Model for XML data: Schemas written in the XML Schema language are converted to Java classes that mimic the schema structure. XML data is represented as objects of these classes. Conversion between textual XML representation and objects is performed by `marshall` and `unmarshall` operations. The mapping from schemas to classes can be customized either via annotations in the schema or in separate binding files.

Type formalism: JAXB relies on the native Java type system. Note that, in contrast to JDOM, this representation is able to obtain static guarantees of certain aspects of validity because the binding reflects many properties of the schemas. Full XML Schema validation is left as a runtime feature.

Annotations: No special annotations are needed in the Java code.

Precision: The approach is pragmatic due to the significant impedance mismatch between XML schema languages and the type system of Java. Using a customized binding, this mismatch can be alleviated, though. Nevertheless, there is no automatic translation from DTD, XML Schema, or the more idealized schema languages that provides precise bindings.

Implementation quality and availability: JAXB has several (even open source) industrial strength implementations.

Data binding frameworks are a commonly used alternative to the DOM/JDOM approach. Still, they constitute a pragmatic approach that cannot provide the static guarantees of conservative frameworks.

4.7 HaXml

If the host language has a more advanced type system, then data binding may be more precise. An example of that is the HaXml [42] system, which uses Haskell as host language. HaXml additionally contains a generic library like JDOM, which we will not consider here.

Language type: HaXml is a data binding framework for Haskell.

Model for XML data: DTDs are converted into algebraic types using a fixed strategy.

Type formalism: HaXml uses the native Haskell types.

Annotations: Type annotations are optional.

Precision: The Haskell type checker is conservative and generally acknowledged to have good precision. For the XML binding, however, the lack of subtyping of algebraic types rejects many natural programs. On the other hand, the presence of polymorphism allows a different class of useful programs to be type checked.

Complexity: The type checker of Haskell is exponential.

Modularity: Modularity is excellent, since Haskell supports separate compilation.

Implementation quality and availability: The Haskell compiler has industrial strength implementations.

In [38] different binding algorithms for Haskell are discussed, enabling more flexible programming styles while trading off full DTD validity.

4.8 Cω

The Cω language (formerly known as Xen) is an extension of C$^\sharp$ that aims at unifying data models for objects, XML, and also databases [4, 31]. This goes a step beyond data binding.

Language type: Cω is an XML-centric GPL based on the C$^\sharp$ language.

Model for XML data: The mutable data values of C$^\sharp$ are extended to include immutable structural sequences, unions, and products on top of objects and simple values. XML trees are then encoded as such values. XML templates may be used as syntactic sugar for the corresponding constructor invocations. A notion of *generalized member access* emulates simple XPath expressions for navigation and deconstruction.

Type formalism: The Cω type system similarly supports structural sequence, union, and product types. There is no support for external types, but the basic features of DTD or XML Schema may be encoded in the type system.

Annotations: Cω requires ubiquitous type annotations, as does C$^\sharp$.

Precision: The Cω type checker appears somewhat restrictive, since the notion of subtyping is not semantically complete: two types whose values are in a subset relation are not necessarily in a subtype relation. This means that many programs will be unfairly rejected by the type checker. For example, an `addrbook` with mandatory `telephone` elements cannot be assigned to a variable expecting an `addrbook` with optional `telephone` elements.

Complexity: The complexity is not stated (but appears to be polynomial given the simple notion of subtyping).

Modularity: The type system is highly modular as that of the underlying C$^\sharp$ language.

Implementation quality and availability: The language is available in a prototype implementation.

Cω solves a more ambitious problem than merely type checked XML transformations, and many of its interesting features arise from the merger of different data models.

4.9 Tree Transducers

XML has a mathematical idealization as ordered labeled trees and schemas may be modeled as regular tree languages. Corresponding to this view, XML transformations may be seen as some notion of *tree transducers*. Two representative examples are TL transformers [30] and k-pebble transducers [32].

Language type: Tree transducers are declarative stand-alone DSLs. Actually, they are generally presented simply as 5-tuples with alphabets and transition functions. The languages are not Turing complete but still capture many central aspects of other transformation languages.

Model for XML data: XML data is immutable. Attributes must be encoded as special nodes, and attribute values and character data are ignored. Construction is performed node by node. Navigation and pattern matching is in the k-pebble approach performed by tree walking and in the TL approach by evaluating formulas in monadic second-order logic.

Type formalism: Types are general regular tree languages.

Annotations: Only the input and output types are specified.

Precision: These classes of tree transducers are particularly interesting, since their type checking problems are decidable. The k-pebble transducers may be viewed as low-level machines, while TL transformers provide a more succinct declarative syntax.

Complexity: The type checking algorithms are hyperexponential.

Modularity: Tree transducers are closed under composition, which provides a simple form of modular type checking.

Implementation quality and availability: The type checking algorithms have not been implemented and are thus pure theoryware. However, non-elementary algorithms on tree automata have previously been seen to be feasible in practice [28].

This work fits into a classical scenario where practical language design and theoretical underpinnings inspire each other. Type checking algorithms for Turing complete languages will become ever more precise and formalisms with decidable type checking will become ever more expressive, but the two will of course never meet.

4.10 XQuery

XQuery is the W3C recommendation for programming transformations of data-centric XML [5] (currently with the status of working draft). As XML trees may been seen to generalize relational databases tables, the XQuery language is designed to generalize the SQL query language.

Language type: XQuery is a Turing complete declarative stand-alone language. Like XDuce, it is also a first-order pure functional language.

Model for XML data: XML data is treated as immutable trees. Future extensions plan to generalize also the update mechanisms of SQL, but the query language itself continues to operate on immutable data. Nodes of trees also have a physical identity, which means that fragments may be either identical or merely equal as labeled trees. A term language is used for constructing values, and XPath is used for deconstruction and pattern matching.

Type formalism: The input and output types are XML Schema instances. Internally, types are tree languages with data values, corresponding to single-type tree grammars. Input/output types are mapped approximately into internal types, and the subtyping is structural (unlike XJ). Most of XML Schema fits into this framework [35].

Annotations: Variables, parameters, and function results have type annotations that by default denote a type containing all values.

Precision: The internal type checker is conservative and based on type rules. Some XQuery constructions are difficult to describe, and the designers acknowledge that some type rules may need to be sharpened in later versions [14]. While XQuery unlike most other transformation languages handle computed element and attribute names, it should be noted that the corresponding type rule must pessimistically assume that any kind of result could arise. The external type checker is strictly speaking pragmatic. The unsoundness arises because XML Schema is only mapped approximately into internal types. To achieve a conservative external type checker, the mapping of the input language should produce an upper approximation and the mapping of the output language a lower approximation. The current mapping apparently meets neither criterion. The practical precision of the type checking algorithm is as yet unknown.

Complexity: The type checking algorithm is exponential.

Modularity: The type checker is in a sense modular, since functions may be type checked separately against their type annotations. However, since the type system lacks analogies of principal type schemes, a single most general choice of type annotations does not exist.

Implementation quality and availability: XQuery is available in several prototype implementations, both commercial and open source. Only a single prototype supports type checking. Industrial strength implementations are being undertaken by several companies.

XQuery will undoubtedly fulfill its goal as an industrial XML standard. In that light, its strong theoretical foundation is a welcome novelty.

4.11 Type Checking XSLT

XSLT 1.0 is the current W3C recommendation for programming transformations of document-centric XML [12]. There is no associated type checker, but in a recent project we have applied the summary graph technology from XACT to create a flow-based external type checker [34]. This tool has been designed to handle a real-life situation, thus the full XSLT 1.0 language is supported.

Language type: XSLT is a declarative stand-alone DSL. It is Turing complete [25], but only in the encoded sense. Even though computed element and attribute names are allowed, there are several XML transformations that cannot be expressed, primarily because XSLT transformations cannot be composed. For example, a transformation that sorts a list of items and alternatingly colors them red and blue cannot be programmed in XSLT.

Model for XML data: XML data is treated as immutable trees. Templates are used as a term language (in a declarative fashion, unlike the template mechanism in XACT). Navigation is performed using XPath.

Type formalism: XSLT 1.0 is itself untyped, but our tool uses summary graphs for internal types. External types are DTDs. The output type may in fact be a DSD2 schema, as in XACT.

Annotations: Our tool works on ordinary XSLT stylesheets, thus only the input and output type must be specified.

Precision: The analysis is conservative. At present, no internal type checks are performed, but we obtain the required information to catch XPath navigation errors and detect dead code. The external type checker has been tested on a dozen scenarios with XSLT files ranging from 35 to 1,353 lines and

DTDs ranging from 8 to 2,278 lines. Most of these examples originate from real-life projects and were culled from the Web. In a total of 3,665 lines of XSLT, the type checker reported 87 errors. Of those, 54 were identified as real problems in the stylesheets, covering a a mixture of misplaced, undefined, or missing elements and attributes, unexpected empty contents, and wrong namespaces. Most are easily found and corrected, but a few seem to indicate serious problems. The 33 false errors fall in two categories. A total of 30 are due to insufficient analysis of string values, which causes problems when attribute values are restricted to `NMTOKEN` in the output schema. A variation of the string analysis presented in [11] may remedy this. The remaining 3 false errors are caused by approximations we introduce and would require sophisticated refinements to avoid. Another important measure of the achieved precision is that the generic identity transformation always type checks.

Complexity: The algorithm is polynomial and appears reasonably efficient in practice. The largest example with 1,353 lines of XSLT, 104 lines of input DTD, and 2,278 lines of output DSD2 schema (for XHTML) ran in 80 seconds on a typical PC (constructing a summary graph with more than 26,000 nodes).

Modularity: The type checker is based on whole-program analysis and thus is not modular.

Implementation quality and availability: XSLT 1.0 has of course many industrial strength implementations. The type checker is implemented as a prototype, which we are currently developing further.

The analysis has several phases. First, we desugar the full XSLT syntax into a smaller core language using only the instructions `apply-templates`, `choose`, `copy-of`, `attribute`, `element`, and `value-of`. The transformation has the property that the type check of the original stylesheet may soundly be performed on the reduced version instead.

Second, we perform a control flow analysis finding for each `apply-templates` instruction the possible target `template` rules. This reduces to checking that two XPath expressions are *compatible* relative to a DTD, which is challenging to solve with the required precision. We use a combination of two heuristic algorithms, partly inspired by a statistical analysis of 200,000 lines of real-life XSLT code written by hundreds of different authors.

Third, a summary graph that soundly represents the possible output documents is constructed based on the control flow graph and the input DTD. The main challenge here is to provide sufficiently precise representations of content sequences in the output language.

Finally, the resulting summary graph is compared to the output schema using the algorithm presented in [10].

There have been other approaches to type checking XSLT stylesheets [40, 2], but our tool is the first working implementation for the full language.

5 Conclusion

Many different programming systems have been proposed for writing transformations of XML data. We have identified a number of aspects by which these systems and their type checking capabilities can be compared. The approaches range from integration of XML into existing languages to development of language extensions or entirely new languages. Some aim for soundness where others are more pragmatic. Additionally, we have presented a brief overview of a novel approach for type checking XSLT 1.0 stylesheets and have shown how this fits into the design space of XML type checkers. An extensive description of this approach is currently in preparation [34].

The variety of approaches indicates that the general problem of integrating XML into programming languages with static guarantees of validity has no canonical solution. Nevertheless, some general observations can be made:

– A fundamental issue seems to be that real-life schema languages are too far from traditional type systems in programming languages.
– The choice of using an immutable representation is common, even in imperative languages. (We have seen the problems that arise with mutability and aliasing).
– It is common to rely on type annotations on all variable declarations. This improves modularity of type checking but is an extra burden on the programmer.
– Many type checkers appear restrictive in the sense that they significantly limit the flexibility of the underlying language. One example is type systems that are not structural; another is rigidity that enforces a programming style where XML trees are constructed purely bottom-up.
– Irrespectively of the type checker, it is important that the language is flexible in supporting common XML transformation scenarios. For example, variants of template mechanisms can be convenient for writing larger fragments of XML data. Also, XPath is widely used for addressing into XML data.
– In many proposals, runtime efficiency is an issue that has not been addressed yet. Also, handling huge amounts of XML data seems problematic for most systems.
– A claim made in many papers is that high theoretical complexity does not appear to be a problem in practice. Nevertheless, it is unclear how well most of the proposed type checking techniques work on real, large scale programming projects.

Overall, the general problem is challenging and far from being definitively solved, and we look forward to seeing the next 700 XML transformation languages and type checking techniques.

Acknowledgments

We thank Claus Brabrand and Christian Kirkegaard for useful comments and inspiring discussions.

References

1. Vidur Apparao et al. Document Object Model (DOM) level 1 specification, October 1998. W3C Recommendation. `http://www.w3.org/TR/REC-DOM-Level-1/`.
2. Philippe Audebaud and Kristoffer Rose. Stylesheet validation. Technical Report RR2000-37, ENS-Lyon, November 2000.
3. Veronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. In *Proc. 8th ACM International Conference on Functional Programming, ICFP '03*, August 2003.
4. Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. Technical report, Microsoft Research, 2004. `http://research.microsoft.com/Comega/`.
5. Scott Boag et al. XQuery 1.0: An XML query language, November 2003. W3C Working Draft. `http://www.w3.org/TR/xquery/`.
6. Ronald Bourret. XML data binding resources, September 2004. `http://www.rpbourret.com/xml/XMLDataBinding.htm`.
7. Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.
8. Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible Markup Language (XML) 1.0 (third edition), February 2004. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.
9. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X '02.
10. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.
11. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.
12. James Clark. XSL transformations (XSLT), November 1999. W3C Recommendation. `http://www.w3.org/TR/xslt`.
13. James Clark and Makoto Murata. RELAX NG specification, December 2001. OASIS. `http://www.oasis-open.org/committees/relax-ng/`.
14. Denise Draper et al. XQuery 1.0 and XPath 2.0 formal semantics, November 2002. W3C Working Draft. `http://www.w3.org/TR/query-semantics/`.
15. Vladimir Gapayev and Benjamin C. Pierce. Regular object types. In *Proc. 17th European Conference on Object-Oriented Programming, ECOOP'03*, volume 2743 of *LNCS*. Springer-Verlag, July 2003.
16. Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. XML goes native: Run-time representations for Xtatic, 2004.
17. Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael Burke, Vivek Sarkar, and Rajesh Bordawekar. XJ: Integration of XML processing into Java. Technical Report RC23007, IBM Research, 2003.
18. Haruo Hosoya. Regular expression filters for XML, January 2004. Presented at Programming Language Technologies for XML, PLAN-X '04.
19. Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints, October 2002. Presented at Programming Language Technologies for XML, PLAN-X '02.

20. Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(4), 2002.
21. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2), 2003.
22. Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 2004. To appear.
23. Jason Hunter and Brett McLaughlin. JDOM, 2004. `http://jdom.org/`.
24. Martin Kempa and Volker Linnemann. On XML objects, October 2002. Presented at Programming Language Technologies for XML, PLAN-X '02.
25. Stephan Kepser. A proof of the Turing-completeness of XSLT and XQuery. Technical report, SFB 441, University of Tübingen, 2002.
26. Christian Kirkegaard, Aske Simon Christensen, and Anders Møller. A runtime system for XML transformations in Java. In *Proc. Second International XML Database Symposium, XSym '04*, volume 3186 of *LNCS*. Springer-Verlag, August 2004.
27. Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.
28. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company.
29. Michael Y. Levin. Compiling regular patterns. In *Proc. 8th ACM SIGPLAN International Conference on Functional Programming, ICFP '03*, August 2003.
30. Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. XML type checking with macro tree transducers. Technical Report TUM-I0407, TU Munich, 2004.
31. Erik Meijer, Wolfram Schulte, and Gavin Bierman. Programming with rectangles, triangles, and circles. In *Proc. XML Conference and Exposition, XML '03*, December 2003.
32. Tova Milo, Dan Suciu, and Victor Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66, February 2002. Special Issue on PODS '00, Elsevier.
33. Anders Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.
34. Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations, 2004. In preparation.
35. Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Proc. Extreme Markup Languages*, August 2001.
36. Frank Neven. Automata, logic, and XML. In *Proc. 16th International Workshop on Computer Science Logic, CSL '02*, September 2002.
37. Sun Microsystems. JAXB, 2004. `http://java.sun.com/xml/jaxb/`.
38. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of Functional Programming*, 12(5):435–468, 2002.
39. Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures, May 2001. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.
40. Akihiko Tozawa. Towards static type checking for XSLT. In *Proc. ACM Symposium on Document Engineering, DocEng '01*, November 2001.

41. Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.
42. Malcolm Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. 4th ACM SIGPLAN International Conference on Functional Programming, ICFP '99*, September 1999.

Recent BRICS Report Series Publications

RS-04-34 Anders Møller and Michael I. Schwartzbach. *The Design Space of Type Checkers for XML Transformation Languages*. December 2004. 21 pp. Appears in Eiter and Libkin, editors, *Database Theory: 10th International Conference*, ICDT '05 Proceedings, LNCS 3363, 2005, pages 17–36.

RS-04-33 Aske Simon Christensen, Christian Kirkegaard, and Anders Møller. *A Runtime System for XML Transformations in Java*. December 2004. 15 pp. Appears in Bellahsene, Milo, Rys, Suciu and Unland, editors, *Database and XML Technologies: Second International XML Database Symposium*, XSym '04 Proceedings, LNCS 3186, 2004, pages 143–157.

RS-04-32 Philipp Gerhardy. *A Quantitative Version of Kirk's Fixed Point Theorem for Asymptotic Contractions*. December 2004. 9 pp.

RS-04-31 Philipp Gerhardy and Ulrich Kohlenbach. *Strongly Uniform Bounds from Semi-Constructive Proofs*. December 2004. 31 pp.

RS-04-30 Olivier Danvy. *From Reduction-Based to Reduction-Free Normalization*. December 2004. 27 pp. Invited talk at the *4th International Workshop on Reduction Strategies in Rewriting and Programming*, WRS 2004 (Aachen, Germany, June 2, 2004). To appear in ENTCS.

RS-04-29 Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. *An Operational Foundation for Delimited Continuations in the CPS Hierarchy*. December 2004. iii+45 pp.

RS-04-28 Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. *A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*. December 2004. 44 pp. Extended version of an article to appear in *Theoretical Computer Science*.

RS-04-27 Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. *On the Adaptiveness of Quicksort*. December 2004. 23 pp. To appear in Demetrescu and Tamassia, editors, *Seventh Workshop on Algorithm Engineering and Experiments*, ALENEX '05 Proceedings, 2005.