BRICS RS-95-17 Koistinen et al.: Design Architectures through Category Constraints

BRICS

Basic Research in Computer Science

Design Architectures through Category Constraints

Jari Koistinen
Nils Klarlund
Michael I. Schwartzbach

BRICS Report Series RS-95-17

ISSN 0909-0878 February 1995

See back inner page for a list of recent publications in the BRICS Report Series. 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@daimi.aau.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Design Architectures through Category Constraints

Jari Koistinen
euajak@eua.ericsson.se
ELLEMTEL Utvecklings Aktiebolag
Box 1505
125 25 Älvsjö, Sweden

Nils Klarlund & Michael I. Schwartzbach
{klarlund,mis}@daimi.aau.dk
BRICS*
Department of Computer Science
University of Aarhus
Ny Munkegade
8000 Aarhus C, Denmark

Keywords: *architectures; language design and implementation; software engineering.*

Abstract

We provide a rigorous and concise formalism for specifying design architectures exterior to the design language. This allows several evolving architectural styles to be supported independently. Such architectural styles are specified in a tailored parse tree logic, which permits automatic support for conformance and consistency. We exemplify these ideas with a small design architecture inspired by real world constraints found in the Ericsson ATM Broadband System.

*Basic Research in Computer Science, Center of the Danish Research Foundation

1 Introduction

For large-scale object-oriented software development, we argue the need for both a general design notation and an independent *architectural style* that captures application domain and platform dependent design decisions. Our goal is to provide a notation for the formal specification of such an architectural style. This will enable the programming environment to provide automatic semantic support for checking that a design conforms to the given architecture.

Experiences at Ellemtel have shown that many important aspects of architectural styles can be expressed through *categories* and *constraints*. In particular, we have been able to express the architectural style for applications built on the Ericsson ATM Broadband System in terms of categories. A category is a semantic specialization of a syntactic entity; for example, object types can be of categories Persistent, Transmittable, or Remote and a method can be of category Asynchronous. Membership of categories will then impose numerous constraints on definitions and uses of these entities. For example, an Asynchronous method in a Remote object type may have only Transmittable arguments, and a Transmittable object cannot be Persistent.

An architectural style will often be intimately connected to an application domain and a specific implementation platform. Thus it is reasonable to expect that several evolving architectural styles must be supported simultaneously. This implies that each style should be specified exterior to the design language and interpreted by a tool in the programming environment.

In this paper we provide a formalism and the necessary algorithms to fulfill this goal. We introduce a *logic of parse trees*, which may conveniently express realistic constraints. Classical results link this logic to tree automata and finally to attribute grammars. Ellemtel and the University of Aarhus are collaborating to implement such a system for the DELOS design language.

Throughout, we exemplify our ideas with a tiny design language and an architectural style that is inspired by the Ericsson ATM Broadband System.

2 Design Architectures

A distinction—although not always well-defined—is often made between languages and notations for analysis, design, and implementation. Objectorientation makes such distinctions even more diffuse since its techniques are expected to be useful in all the three main phases of software development.

We believe a design language should help developers express important design decisions on a higher level of abstraction than programming languages; for example, it should not be just a graphical programming language. It should make it possible to coordinate the design of different views of a system in a way that transcends type cliches of the programming language. Thus the design language allows design decisions to be made explicit, thereby reducing the available choices during implementation. Furthermore, we believe a design language should support the consideration of architectural styles and implementation platform constraints.

The design language DELOS for large object-oriented distributed systems is under development at Ellemtel [9]. The focus on design aspects of software development implies that DELOS, while covering more areas than implementation languages, excludes some common programming constructs.

Briefly, the reasons for developing a new design language can be summarized as follows.

- DELOS aims to cover more aspects than existing object-oriented design notations such as [2, 11, 16, 5, 13].

- DELOS aims at a level of formalization not currently available in other object-oriented design notations.

- The telecommunications domain requires specialized abstractions that are reflecting the application domain and implementation platforms.

DELOS consists of three parts formally integrated into one unified language.

- OM (Object-Modeling): a type and interface definition language based on concepts usually considered central to object-oriented languages [15].

- SM (Structure-Modeling): coarse-grained, high-level modularization of collections of object types and hierarchical structuring of these modules [10].

- DM (Distribution-Modeling): design of process structures, distribution of processes, distribution of persistent data, etc.

Some general concepts and constraints are made concrete in DELOS through ordinary language constructs and their semantics. Other more arbitrary concepts and restrictions mirror design decisions for the application domains and platforms where DELOS is currently used. We have therefore chosen to let DELOS be a core language [9], largely independent of the application area, and to provide an orthogonal formalism for describing system wide design decisions and constraints constituting an architectural style.

Currently, DELOS is used for designing applications on the Ericsson ATM Broadband System. The specializations and restrictions for this platform are described separately in a platform and architecture specific manual [8]. We foresee that as the Ericsson ATM Broadband System platform evolves, the constraints and architectural concepts will change. This will result in different sets of categories for different versions of the platform. In addition, DELOS aims to be a general purpose design language tailorable for different styles and platforms. It is therefore of significant importance that categories can be loaded dynamically by the DELOS design tools.

Architectures and Architectural Styles

Architectures are concerned with how program elements are chosen and composed. We agree with Buxton and McDermid [4] in their statement that an architecture:

- "...defines the structure of the system, its functionality, etc., in such way that the system can be built. "
- "...provides most of the information necessary to enable the remainder of the development process to be organized and planned."

Structures represent particular construction elements that are compositions of other constructions elements such as classes, large-grained modules, functions, etc. A structure represents a physical system and has an architecture.

Architectural styles [12, 6] capture important design decisions about construction elements and their relations based on some specific application domain or on the possibilities and limitations of a technology. An architectural style can for example describe how to build distributed systems in the domain of telecommunication systems using a specific switching technique and operating system; or how to structure predicates of logic programs.

An architecture is more specific than an architectural style, in the sense that it represents one formal arrangement of construction elements. In contrast, an architectural style captures characteristics that are common for many architectures.

We represent an architectural style for static structures through a set of *categories*. A category characterizes the entities and structure patterns used for systems with an architecture conforming to a specific architectural style. Typically, we identify categories of object types that characterize architectures in a particular application domain and on a particular platform.

For a very simple example, assume we have a need for transmitting objects between processes in a distributed system. We may have noted that marshalling and type-checking is significantly eased and speeded up if we do not need to consider inheritance and polymorphism for transmitted objects. Furthermore, we wish to use the same techniques and basic design for data transmitted between processes in all systems on this particular implementation platform. We therefore decide that the type of a transmitted object must belong to a certain category restricting the use of inheritance.

This is certainly an important design decision which will affect all systems built on this platform. The definition of the category might, however, change as better technologies for marshalling emerges. Using categories such a change is not a problem since the design language itself is not directly affected. Furthermore, tools supporting categories are more generic with respect to such changes than tools built without category support.

Architectural styles are defined by system architects and used by the developers when they design systems. An architectural style is commonly used in several systems within the same application area and on the same implementation platform.

The emphasis on architectural styles and their formalization makes it easier to communicate designs and architectures but also to ensure that all designs conform to the basic decisions made by the system architects. In the development of large systems they will also help to enforce a common architectural style on the system as a whole.

Benefits of Formalized Categories

At Ellemtel, categories are with significant success used to describe styles at both the coarse-grained level (DELOS/SM) and at the level of object-types

(DELOS/OM). Although the concept of architectural style is useful in its own right, our experiences have shown that the real advantages come when an architectural style can be formalized *i.e.*, when we have a *category definition language*.

In order to formalize categories we need two things: a way of describing that a definition in a design belongs to a certain category; and a way of describing the constraints imposed by a category.

In DELOS we can specify explicitly in definitions—such as of object types—that they belong to certain categories. Each definition belonging to a category must conform to the restrictions it imposes. In Figure 1 we outline the simple example mentioned earlier, where the types of objects transmittable between processes are not allowed to inherit. The name of the category is Transmittable and the object type Subscriber belongs to that category. The category definition language will be described in more detail later in this paper.

```
category Transmittable is
    "no inheritance clause is allowed"
end

object type Transmittable : Subscriber is
    attributes
        id : integer;
        nr : bcdstring;
    methods
        ...
end
```

Figure 1: A category Transmittable and an object type Subscriber of that category.

The formal description of design constraints is advantageous from two different points of view. Firstly, constraints can be unambiguously described and conformance to them can be automatically verified during the design activity. Secondly, constraints can be used during translation from design to implementation in order to map definitions of different categories to specific concepts and constructs of the implementation platform.

From a design perspective, the main advantages of describing architectural constraints in terms of categories are that:

- the architectural concepts gain formal descriptions that can be understood and discussed more easily;
- the constraints on syntax and semantics of architectural elements can be described in a systematic and formal way;
- the designs can be discussed using categories as a common architectural framework; and
- concepts used in the development process can be formalized in terms of categories at the design language level.

From a language tool perspective, formal category definitions can be used to:

- restrict the usage of language constructs for certain architectural elements during the actual modeling activity;
- omit implementing translations for forbidden constructs, which eases the adaptation of design language tools to new architectures and platform; and
- recognize categories during translation and map language concepts into platform specific concepts and constructs automatically.

At least two issues in the use of architectural styles will benefit from semantic support from the programming environment.

- Conformance: given a specific design, does it conform to the architectural style?
- Consistency: given an architectural style, does it contain contradictory requirements on designs?

Conformance will be checked continuously during the design phase. Consistency is only checked once for each architectural style, but this may avoid severe problems during later design projects. If a style is defined through hundreds of individual constraints, then it is all too easy to include two that contradict each other. The formalism we shall propose meets the following stringent requirements.

- The architectural style is dynamically configurable through definitions in a separate `.arch`-file.
- It uses a logical language to express constraints on categories.
- This language is sufficiently expressive to capture existing styles.
- Conformance can be verified efficiently in linear time.
- Consistency of an architectural style is decidable.

In the following sections of this paper we will abstract from the concrete application of categories in Delos. Instead we will describe how the concept of categories can be formalized while obtaining the expressive power and other characteristics that we believe are necessary.

```
Module      ::= module ModuleName
                  { Module | ObjectType } *
                end
ObjectType  ::= object type TypeName { inherits TypeName } is
                  attributes Attribute *
                  methods Method *
                end
Attribute   ::= AttName : Type ;
Type        ::= TypeName | integer | boolean | bcdstring
Method      ::= method ( Argument * )
                  Statement *
                end
Argument    ::= ArgName : Type ;
Statement   ::= new TypeName |
                spawn TypeName |
                self |
                Call |
                ⋮
```

Figure 2: Syntax of a Tiny Design Language.

3 Categories and Constraints

To exemplify our ideas, we first define a tiny design language and an example of an architectural style expressed through categories and constraints.

The design language is quite ordinary, supporting simply modules, object types, attributes and methods. Its context-free syntax is shown in Figure 2.

We shall consider an architectural style, inspired by (a tiny subset of) Ericsson ATM Broadband System, that introduces the following different categories of object types.

- Abstract: is used only for inheritance; does not permit instantiation.
- Concrete: the complement of Abstract.
- Transmittable: instances can be transmitted between processes.
- Persistent: instances can be saved in a persistent store.
- Remote: instances reside in a separate process.
- Plain: can be mapped to classes in most OO languages.
- Base: encapsulates base functions.
- Extension: encapsulates extensions of base functions.

Most of these correspond more or less to everyday intuitive concepts, but Base and Extension are quite special to telecommunication applications. Object types of categories like Remote and Persistent have many constraints and are also treated quite differently from for example Plain during the translation to implementation code.

Categories are assigned to object types explicitly by the programmer, as indicated in Figure 1. An object type may have several categories at once, but certain restrictions must be obeyed. For example, an object type must be either Abstract or Concrete, and an object type cannot according to our constraints be both Transmittable and Persistent.

Architectural categories can be defined for every syntactic category. For example, a module can be of category Swilib, which means that it contains only Abstract object types. Also, an attribute of a persistent object type that is of category Asynchronous will not be protected from simultaneous access by clients.

Not all category annotations are provided explicitly by the programmer. Some are supplied automatically by the compiler in a generic manner, as follows.

- Firstly, statements are automatically assigned categories corresponding to the coarse grouping in Figure 2, *i.e.*, new, spawn, self, call, etc.
- Secondly, categories are by default inherited from supertypes.
- Finally, an occurrence of a TypeName is annotated with all the categories of its defining object type.

Note that these synthesized annotations are independent of the specific architecture. The architectural style must then impose the required constraints on such designs with complete category annotations. Already in this simple example there are numerous constraints that must obviously hold.

1. An object type is either Concrete or Abstract.
2. An object type cannot be both Transmittable and Persistent.
3. An Asynchronous method in a Remote object type may have only Transmittable arguments.
4. A Transmittable object type cannot have supertypes.
5. A new statement cannot use an Abstract object type.
6. A Base object type has exactly one method.
7. A spawn statement can only use a Base object type.
8. An Extension object type cannot have subtypes.
9. A Swilib module has exclusively Abstract object types.

Figure 3: Examples of Constraints on Categories.

In Figure 3 we mention just a few that we shall later formalize. A complete and meaningful architectural style for this tiny design language would require dozens of individual constraints. The Ericsson ATM Broadband System architecture for the DELOS language requires hundreds of similar or more complicated constraints.

If we were only to encounter a single monolithic architectural style then it would make sense to manually extend the programming environment to provide semantic support by verifying that all these constraints are valid in proposed designs. However, as we have argued, we want simultaneously to handle several evolving architectural styles. The obvious solution is to specify all the different categories and constraints in a separate `.arch`-file, which is then dynamically interpreted by some tool in the programming environment.

This leaves us with the challenges of defining a formal language for expressing such constraints and building the corresponding tool.

```
object type Abstract,Transmittable: T is
   attributes i: integer; t: T;
   methods
end
```

```
ObjectType: Abstract,Transmittable
  ├── object type
  ├── TypeName
  │     └── T
  ├── is
  ├── attributes
  ├── Attribute
  │     ├── AttName
  │     │     └── i
  │     └── Type
  │           └── integer
  ├── Attribute
  │     ├── AttName
  │     │     └── t
  │     └── Type
  │           └── TypeName: Abstract,Transmittable
  │                 └── T
  ├── methods
  └── end
```

Figure 4: A Tiny Program and its Parse Tree.

4 Formal Specifications

The category constraints that are practically useful can all be viewed as structural restrictions on parse trees extended with category annotations on nodes. A tiny object type and its parse tree are shown in Figure 4. We could formalize such constraints as (complicated) grammars. However, the constraints have traditionally been described in a semi-formal fragment of predicate logic, which has proved to be a compact and highly intuitive notation.

It seems that we have a dilemma. On one hand, predicate logic is convenient but not decidable. On the other hand, while grammars are decidable they are also cumbersome and even simple properties may require huge descriptions. Fortunately, the literature describes a formalism that is easily adapted to our purposes [14].

$$\begin{array}{rcl} F & ::= & \nu \leq \nu \ \mid \ \nu \rightarrow \nu \ \mid \ \mathsf{t}(\nu) \ \mid \ \mathsf{N}(\nu) \ \mid \ \mathsf{C}(\nu) \ \mid \\ & & \neg F \ \mid \ F \vee F \ \mid \ F \wedge F \ \mid \ \forall \nu : F \ \mid \ \exists \nu : F \\ \nu & ::= & \alpha \ \mid \ \beta \ \mid \ \gamma \ \mid \ \ldots \end{array}$$

Figure 5: Syntax of the Parse Tree Logic.

The first-order logic shown in Figure 5 consists of formulas on parse trees. That is, each formula F is either true or false of a given tree. Variables range over the nodes of the parse tree. The connectives $(\neg, \vee, \wedge, \forall, \exists)$ have the usual semantics. The basic predicates are defined as follows. The predicate $\alpha \leq \beta$ holds if the node α is on a path from the root to β; $\alpha \rightarrow \beta$ holds if β is a right sibling of α; $\mathsf{t}(\alpha)$ holds if α is labeled with the terminal symbol t; $\mathsf{N}(\alpha)$ holds if α is labeled with the non-terminal N; and $\mathsf{C}(\alpha)$ holds if α is annotated with the category C.

This logic has been chosen very carefully to meet our needs. In the following section we shall see that it is efficiently decidable. For now we shall argue that it is sufficiently expressive to capture realistic category constraints.

Basic examples of formulas can be seen in Figure 6, which shows some useful abbreviations that we shall use below. The definitions of implication $(\Rightarrow)$, exclusive-or $(\oplus)$, equality $(=)$, inequality $(\neq)$, and strict order $(<)$ are straightforward. The abbreviation $\alpha \lhd \beta$ means that β is a node *immediately*

$$
\begin{array}{rcl}
F \Rightarrow G & \equiv & \neg F \vee G \\
F \oplus G & \equiv & (F \vee G) \wedge (\neg F \vee \neg G) \\
\alpha = \beta & \equiv & \alpha \leq \beta \wedge \beta \leq \alpha \\
\alpha \neq \beta & \equiv & \neg\, \alpha = \beta \\
\alpha < \beta & \equiv & \alpha \leq \beta \wedge \alpha \neq \beta \\
\alpha \lhd \beta & \equiv & \alpha < \beta \wedge \neg\exists\gamma : (\alpha < \gamma) \wedge (\gamma < \beta) \\
\exists!\alpha : F(\alpha) & \equiv & \exists\alpha : F(\alpha) \wedge \forall\alpha,\beta : F(\alpha) \wedge F(\beta) \Rightarrow \alpha = \beta \\
\mathsf{anArgument}(\alpha,\beta) & \equiv & \alpha \lhd \beta \wedge \textsc{Argument}(\beta) \\
\mathsf{aMethod}(\alpha,\beta) & \equiv & \alpha \lhd \beta \wedge \textsc{Method}(\beta) \\
\mathsf{theType}(\alpha,\beta) & \equiv & \alpha \lhd \beta \wedge \textsc{Type}(\beta) \\
\mathsf{theTypeName}(\alpha,\beta) & \equiv & \alpha \lhd \beta \wedge \textsc{TypeName}(\beta) \\
\mathsf{theSuperType}(\alpha,\beta) & \equiv & \exists\gamma : \alpha \lhd \gamma \wedge \textsf{\textbf{inherits}}(\gamma) \wedge \gamma \rightarrow \beta \wedge \textsc{TypeName}(\beta)
\end{array}
$$

Figure 6: Some Useful Abbreviations.

below α. The derived quantification $\exists!$ indicates the existence of exactly one node. The latter formulas are used to indicate nodes in relative position of each other according to production in the grammar. For example, the formula $\mathsf{theSuperType}(\alpha, \beta)$ states that β is the parse tree node denoting the name of the super type of the object type α. This last formula is a bit messy—let us spell out what it means: find a node γ immediately below α that is labeled with the terminal symbol **inherits**; the node β we want is a right sibling of γ labeled with the nonterminal symbol TYPENAME.

As a larger example of the expressiveness of our logic, note that it is a simple task to give a formula $\mathsf{Parse}(\alpha)$ that holds *iff* α is the root of a valid parse tree according to a given context-free grammar.

Figure 7 shows the complete formalization of the category constraints from Figure 3. It should be apparent that our formalism is fairly intuitive and quite succinct. Many of the formulas even resemble earlier semi-formal descriptions of such constraints. Of course, an obvious shorthand would be to define that free logical variables are implicitly universally quantified.

1. $\forall\alpha : \textsc{ObjectType}(\alpha) \Rightarrow \mathsf{Abstract}(\alpha)\ \oplus\ \mathsf{Concrete}(\alpha)$

2. $\forall\alpha : \textsc{ObjectType}(\alpha) \Rightarrow \neg\,(\mathsf{Transmittable}(\alpha)\ \wedge\ \mathsf{Persistent}(\alpha))$

3. $\forall\alpha,\beta,\gamma,\delta,\epsilon :\ \mathsf{Remote}(\alpha)\ \wedge\ \mathsf{aMethod}(\alpha,\beta) \wedge$
   $\mathsf{Asynchronous}(\beta)\ \wedge\ \mathsf{anArgument}(\beta,\gamma) \wedge$
   $\mathsf{theType}(\gamma,\delta) \wedge \mathsf{theTypeName}(\delta,\epsilon) \Rightarrow \mathsf{Transmittable}(\epsilon)$

4. $\forall\alpha :\ \textsc{ObjectType}(\alpha)\ \wedge\ \mathsf{Transmittable}(\alpha)\ \Rightarrow$
   $\neg\exists\beta :\ \mathsf{theSuperType}(\alpha,\beta)$

5. $\forall\alpha,\beta :\ \mathsf{new}(\alpha)\ \wedge\ \mathsf{theTypeName}(\alpha,\beta)\ \Rightarrow\ \neg\,\mathsf{Abstract}(\beta)$

6. $\forall\alpha,\beta :\ \mathsf{Base}(\alpha)\ \Rightarrow\ \exists!\beta :\ \mathsf{aMethod}(\alpha,\beta)$

7. $\forall\alpha,\beta : \mathsf{spawn}(\alpha)\ \wedge\ \mathsf{theTypeName}(\alpha,\beta)\ \Rightarrow\ \mathsf{Base}(\beta)$

8. $\forall\alpha,\beta :\ \textsc{ObjectType}(\alpha)\ \wedge\ \mathsf{theSuperType}(\alpha,\beta)\ \Rightarrow\ \neg\,\mathsf{Extension}(\beta)$

9. $\forall\alpha,\beta : \mathsf{Swilib}(\alpha)\ \wedge\ \alpha \leq \beta\ \wedge\ \textsc{ObjectType}(\beta)\ \Rightarrow\ \mathsf{Abstract}(\beta)$

Figure 7: Formalizations of Example Constraints.

5 Algorithms for Semantic Support

In this section, we argue that conformance can be efficiently supported in practice and that consistency is decidable.

This is perhaps surprising since deciding the truth-status of a formula in tree logics like ours is known to require *non-elementary time*, see [14]; thus, it may happen that a formula of length n requires time given by a stack of exponential functions whose height is proportional to n. This grim lower bound is also an indication of the expressive power of the logic: complicated properties can be expressed very concisely.

Note however that there are many properties of labeled trees that cannot be expressed in such tree logics. For example, we cannot represent the symbol table corresponding to the parse tree in the logic. Hence the automatic synthesis of category annotations provided by the compiler is quite necessary.

The Decision Procedure

To decide the truth-value of a given formula, we first calculate for each subformula F a tree automaton A_F that recognizes the set of interpretations satisfying F.

This construction is performed inductively in the syntax of formulas. Atomic formulas correspond to automata with only a couple of states; $F \wedge G$ and $F \vee G$ are translated by cross product constructions of the automata corresponding to F and G; negation $\neg F$ switches final and non-final states; and existential quantification corresponds to a projection on the alphabet and a determinization of the resulting non-deterministic automaton.

The interpretation of an open formula such as $\alpha \leq \beta \wedge \beta \rightarrow \gamma$ extends the alphabet of the automaton to include the values of the free variables α, β, and γ. The set of labeled trees corresponding to interpretations that make this particular formula true can then be recognized by a tree automaton with approximately ten states.

Note that these automata need only be constructed once for a given design architecture in a tool generation phase. The automata are then used to verify that a parse tree respects the corresponding formula by running the automata on the tree. If each automata finishes in an accept state, then the parse tree satisfies the formulas. Thus, given a fixed architectural styles, conformance checking requires only time linear in the size of the parse tree. The entire process is sketched in Figure 8.

When a tree automaton has been constructed, it is easily translated into an attribute grammar formalism, where the synthesized attributes hold the states of the automaton. This is of significant practical importance, since many programming environments already support attribute grammars.

In contrast, consistency is verified by a single (huge) formula with the overall structure:

$$\mathsf{Parse} \Rightarrow \exists labeling : \bigwedge_i F_i$$

where Parse is a subformula denoting that the underlying tree is a valid parse tree; $\exists labeling$ indicates that the existence of a category labeling; and F_i denotes the i'th category constraint. Since this formula involves an explicit product construction of the automata corresponding to the individual constraints, we will encounter a potential state space explosion. It is possible to avoid constructing the product of all automata if some constraints are

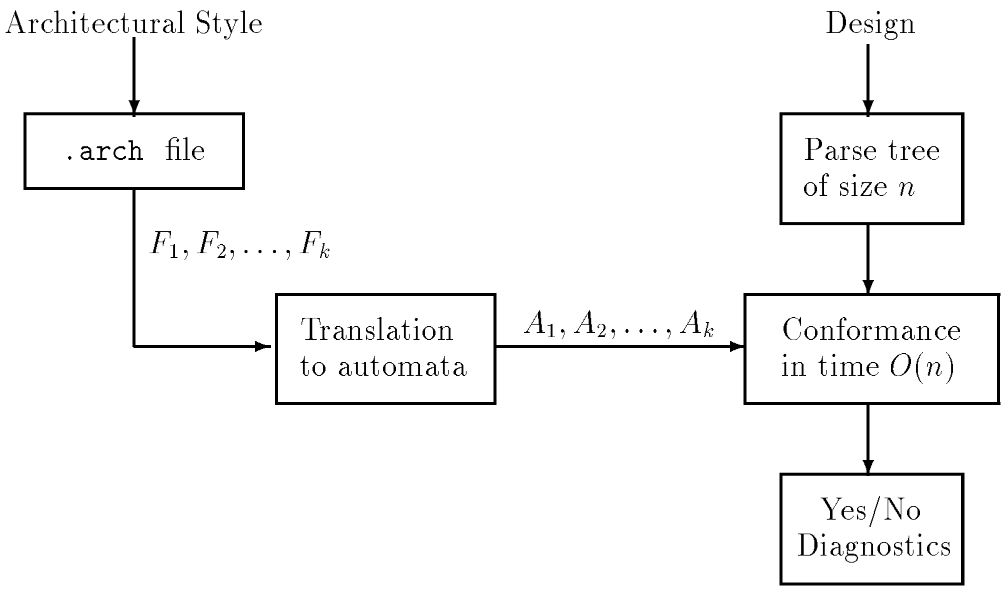

Figure 8: Generating and Using a Conformance Checker.

independent of others, which is certainly likely to be the case. But only experiments can show whether consistency can be checked in practice.

Implementation

At the University of Aarhus, the decision procedure has been implemented for the special case of linear trees, *i.e.* strings. The use of Binary Decision Diagrams [3] and special algorithms yield high performance; that is, formulas consisting of a few hundred symbols are processed in a fraction of a second and large formulas of a few hundred thousand characters are processed in minutes [1, 7]. Of course, these figures assume that the resulting automata are themselves not explosively big, say no more than a million BDD nodes for formulas of size 10^5.

Fortunately, simple individual formulas in tree logic like the ones we have shown here will result in considerably smaller automata even if tree automata inherently tend to be bigger than automata on strings. Nothing we have shown in this paper need require automata with more than a few dozen states.

Also, since the computed automata are minimal, no procedure that actually verifies the formulas can be essentially smaller. Thus, we are quite confident that these ideas will scale to realistic sizes. An ongoing collaboration between the University of Aarhus and Ellemtel will implement the full parse tree logic and integrate it into the DELOS programming environment. Figure 9 shows the possible contents of a `.arch`-file which would be the input to our tool. It contains constraints 2, 4 and 9 from Figure 7 coated in a layer of syntactic sugar. Currently the full collection of constraints for the Ericsson ATM Broadband System is being translated into our formalism.

```
category Transmittable(a: ObjectType) is
   not Persistent(a);
   not (exists b: theSuperType(a,b));
   ...
end

category Swilib(a: Module) is
   (a <= b) and ObjectType(b) implies Abstract(b);
   ...
end
```

Figure 9: Possible Contents of a `.arch`-file.

6 Conclusion

Our logic of parse trees is an intuitive and succinct notation for the intricate syntactic restrictions imposed by architectural styles. We are currently expressing the full Ericsson ATM Broadband System architecture in this notation.

Our approach has some inherent limitations. Some reasonable constraints cannot be expressed in our logic. However, we have made a very appealing compromise between expressiveness and feasibility. It is unlikely that automatic semantic support will be possible for significantly stronger notations.

With the promise of efficient algorithms, we are confident that our approach will survive the perilous transition from theory to practice.

References

[1] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. Technical Report RS-95-7, BRICS, 1995.

[2] Grady Booch. *Object-oriented analysis and design with applications.* The Benjamin/Cummings publishing company, 1993.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, August 1986.

[4] John Buxton and John A McDermid. Architectural design. In John A McDermid, editor, *Software Engineer's Reference Book*, section 17. Butterworth-Heinemann Ltd., 1991.

[5] Peter Coad and Edward Yourdon. *Object-Oriented Design.* Yourdon-Press, first edition, 1991.

[6] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT*, (12), December 1994.

[7] N. Klarlund, M. Nielsen, and K. Sunesen. Abstraction mappings. Technical report, BRICS, 1995. In preparation.

[8] Jari Koistinen, Eui-Suk Chung, Mats Svensson, and Martin Boström. Delos 2.1 for AXE: AXE specific categories. Ellemtel Telecommunication Systems Laboratories, January 1995.

[9] Jari Koistinen, Eui-Suk Chung, Mats Svensson, and Martin Boström. Delos 2.1 language description: Basic Delos. Ellemtel Telecommunication Systems Laboratories, January 1995.

[10] Jari Koistinen and Einar Wennmyr. Delos/SM: A language for structuring of coarse-grained modules, spec ifications, and interfaces. In *Proceedings of XV International Switching Symposium*, April 1995. To be published.

[11] James Martin and James J. Odell. *Object-Oriented Analysis & Design.* Prentice-Hall, 1992.

[12] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM Software Engineering Notes*, 17(4), October 1992.

[13] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[14] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. MIT Press/Elsevier, 1990.

[15] Peter Wegner. Dimensions of object-based language design. In *OOPSLA'87, Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987.

[16] Rebecca J. Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice-Hall, 1990.

Recent Publications in the BRICS Report Series

RS-95-17 **Jari Koistinen, Nils Klarlund, and Michael I. Schwartzbach.** ***Design Architectures through Category Constraints*****. February 1995. 19 pp.**

RS-95-16 **Dany Breslauer and Ramesh Hariharan.** ***Optimal Parallel Construction of Minimal Suffix and Factor Automata*****. February 1995. 9 pp.**

RS-95-15 **Devdatt P. Dubhashi, Grammati E. Pantziou, Paul G. Spirakis, and Christos D. Zaroliagis.** ***The Fourth Moment in Luby's Distribution*****. February 1995. 10 pp.**

RS-95-14 **Devdatt P. Dubhashi.** ***Inclusion–Exclusion*****(3)** ***Implies Inclusion–Exclusion*****(n). February 1995. 6 pp.**

RS-95-13 **Torben Braüner.** ***The Girard Translation Extended with Recursion*****. 1995. Full version of paper to appear in Proceedings of CSL '94, LNCS.**

RS-95-12 **Gerth Stølting Brodal.** ***Fast Meldable Priority Queues*****. February 1995. 12 pp.**

RS-95-11 **Alberto Apostolico and Dany Breslauer.** ***An Optimal*** **$O(\log \log n)$** ***Time Parallel Algorithm for Detecting all Squares in a String*****. February 1995. 18 pp. To appear in SIAM Journal on Computing.**

RS-95-10 **Dany Breslauer and Devdatt P. Dubhashi.** ***Transforming Comparison Model Lower Bounds to the Parallel-Random-Access-Machine*****. February 1995. 11 pp.**

RS-95-9 **Lars R. Knudsen.** ***Partial and Higher Order Differentials and Applications to the DES*****. February 1995. 24 pp.**

RS-95-8 **Ole I. Hougaard, Michael I. Schwartzbach, and Hosein Askari.** ***Type Inference of Turbo Pascal*****. February 1995. 19 pp.**

RS-95-7 **David A. Basin and Nils Klarlund.** ***Hardware Verification using Monadic Second-Order Logic*****. January 1995. 13 pp.**

RS-95-6 **Igor Walukiewicz.** ***A Complete Deductive System for the*** **μ*****-Calculus*****. January 1995. 39 pp.**