A comparison of metacompilation approaches to implementing Modelica

David Broman, Peter Fritzson,
Görel Hedin, Johan Åkesson

Report 97, 2011
Department of Computer Science
Lund University

ISSN 1401-1200
Report 97, 2011
LU-CS-TR:2011-248

Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

A comparison of metacompilation approaches to implementing Modelica

David Broman and Peter Fritzson
Dept. of Computer Science, Linköping University, Sweden

Görel Hedin
Dept. of Computer Science, Lund University, Sweden

Johan Åkesson
Dept. of Automatic Control, Lund University, Sweden

October 28, 2011

Abstract

Operational semantics and attribute grammars are examples of formalisms that can be used for generating compilers. We are interested in finding similarities and differences in how these approaches are applied to complex languages, and for generating compilers of such maturity that they have users in industry.

As a specific case, we present a comparative analysis of two compilers for Modelica, a language for physical modeling, and which contains numerous compilation challenges. The two compilers are OpenModelica, which is based on big-step operational semantics, and JModelica.org, which is based on reference attribute grammars.

1 Introduction

Computer languages constitute a key way of supporting problem solving in different domains, but require large investments for building and supporting tools like compilers, [23, 14]. In particular, if the domain-specific language is sufficiently complex, the full arsenal of compilation and program analysis techniques may be needed, and it is not sufficient to rely on light-weight approaches like embedded domain-specific languages [10].

One technique for bringing down the tool development cost is to develop *metacompilers*, i.e., languages for the compiler domain, allowing compilers and related tools to be generated from high level specifications. Over the years, there have been numerous such attempts, based on different formalisms such as operational semantics, denotational semantics, attribute grammars, and algebraic term rewriting. While such metacompilation tools are still not mainstream, there are examples of successful applications leading to tools used outside the research labs, see, e.g., [14].

The goal of this paper is to provide insight into how such approaches can be applied to complex domain-specific languages in industrial use. How are key

compilation problems solved in the different approaches? What benefits are achieved concerning quality aspects, for example, modularity and extensibility?

As a case study, we have chosen to study compilers for the language *Modelica*[1], a language for modeling and simulation of physical systems, based on hybrid differential-algebraic equations (DAEs). There are a number of commercial and free implementations of the language, two of which are implemented using metacompilation: *OpenModelica*[2] which is implemented using the RML metacompiler [16], based on big-step operational semantics (BSOS) [12]; and *JModelica.org*[3] which is implemented using the JastAdd metacompiler [9], based on reference attribute grammars (RAGs) [8]. Although both approaches use high-level declarative formalisms, they are quite different: operational semantics is closely related to logic and functional programming, whereas reference attribute grammars are closer to object-oriented programming. Both compilers are of such maturity that they are used in industrial projects.

The main contribution of this paper is the comparative analysis of mature applications of the two approaches (RML BSOS and JastAdd RAGs) to a complex language (Modelica).

The rest of this paper is structured as follows. Section 2 introduces the Modelica language, and the metacompilation approaches of big-step operational semantics and reference attribute grammars. Section 3 identifies a number of key problems that a compiler for Modelica needs to solve. Section 4 explains how these problems are solved using the two metacompilation approaches. Section 5 provides a concluding discussion of similarities and differences between the two approaches.

2 Background

This section gives some background on the Modelica language, and on the two declarative compilation approaches used: big-step operational semantics (BSOS) and reference attribute grammars (RAGs).

2.1 The Modelica Language

The Modelica language has evolved since the mid 90s from the field of continuous-time simulation of physical systems. The language design has been, and still is, subject to significant developments to accommodate support for increasingly complex models. The development process is influenced both by modeling practitioners and by language and compiler experts.

Modelica is object-oriented, strongly typed, and offers abstractions like classes, inheritance, generics, and components, which are useful for building reusable model libraries. The language allows users to express physical relations using mathematical equations, rather than assignments. Model components can be connected, acausally, to each other in order to create complex composite models from simpler component models. Component models may come from different physical domains, including electronics, mechanics and thermodynamics. In addition to modeling of continuous-time behavior, such as conservation of mass

[1] `http://www.modelica.org`
[2] `http://www.openmodelica.org`
[3] `http://www.jmodelica.org`

and energy, Modelica supports modeling of discrete behavior. This feature is useful for modeling of physical phenomena like friction or in modeling of control systems. Modelica supports both textual modeling and visual drag and drop modeling where components are connected graphically.

Modelica is designed to facilitate *simulation* of large-scale dynamic models. In the following very simple example, the model `Newton` describes the position `pos` and the velocity `vel` of an object with mass `m` under the force `f`. The equations in the model describe the well-known Newtonian law relating position, velocity, mass, and force, using time-derivatives (the `der` function).

```
model Newton
  parameter Real m(unit="kg")=1;
  Real pos(unit="m",start=1);
  Real vel(unit="m/s",start=0);
  input Real f(unit="N");
equation
  der(pos) = vel;
  m * der(vel) = f;
end Newton;
```

In simulation, the model is used to simulate the behavior of variables like `pos` and `vel` as a function of time. Typically, models consist of many objects, of different model classes, connected together. A complete model may contain tens of thousands of individual equations.

2.2 Big-step Operational Semantics

The use of *operational semantics* as a formal specification formalism was first advocated by Plotkin in 1981 [18], but the underlying ideas date back to the early 1970s. Plot-kin's work emphasizes on what is today known as *small-step operational semantics* (also known as structural operational semantics), which is a syntax-directed formalism for describing an abstract machine with less complex machinery. Small-step operational semantics has been extensively used in the past decades both for describing semantics of programming languages and for proving properties such as type safety [17].

Another variant of operational semantics, today known as *big-step operational semantics* (originally called natural semantics), was first developed by Kahn in 1987 [12]. A definition in this formalism consists of an unordered collection of rules, known as a rule set. Each rule can either be an axiom or an inference rule. For example, the inference rule

$$\frac{E \vdash e_1 \Rightarrow \texttt{true} \qquad E \vdash e_2 \Rightarrow v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v} \tag{1}$$

defines the dynamic semantics of the `true`-branch of a Modelica `if`-expression, where E is the environment, e_1 the guard expression, e_2 the `true`-branch, and e_3 the `false`-branch. The rules can be read clock-wise as follows: If expression e_1 in environment E is evaluated to `true` then evaluate e_2 to value v and return v. The unordered collection of formulas above the line is called the *premises* of the rule and the formula below the line is called the *conclusion*.

Several specification languages and compilers for operational semantics exist, including *Relational Meta-Language (RML)* [16], Maude [21], and TYPOL [5]. RML has been used to implement OpenModelica [7], an open-source implemen-

tation of the Modelica language and is the operational semantics specification language discussed in this paper.

RML is primarily used for defining executable big-step operational semantics. Rules are defined as axioms or inference rules and are grouped into collections of rules. For performance reasons and to avoid verbose specifications, RML rules are order dependent and evaluated in more functional style than the relational style originally proposed by Kahn. However, compared to functional languages, such as OCaml or Haskell, RML includes language constructs inspired from logic programming. The most notable such construct is *backtracking* of rules, i.e., if a premise fails, backtracking of the rule occurs and the next rule in the collection of rules is executed.

2.3 Reference Attribute Grammars

Attribute Grammars (AGs) were introduced by Knuth in 1968 to define the semantics of programming languages [13]. In AGs, syntax tree nodes are decorated with *attributes* that are defined declaratively using *semantic functions* (also called *equations*). The attributes are classified as either *synthesized* or *inherited*, depending on if their defining equation is located in the same node as the attribute, or in a parent node. The introduction of AGs triggered a large amount of both practical and theoretical research, and many suggested extensions [15].

One extension to AGs is that of *reference attributes* [8]. Here, the syntax tree is viewed as a tree of objects and attributes are allowed to be references to such objects. The reference attributes can be dereferenced to access non-local properties, for example properties of declared variables or inherited classes. As an example, consider a grammar where `Use` is a subclass of `Exp` (the grammar is modelled by a class hierarchy). The following fragment declares that (1) each `Exp` node has a synthesized reference attribute `type` of class `Type`, and that (2) the `type` of a `Use` node is the same as the type of its declaration, where `decl` is a reference attribute, referring to the appropriate declaration node.

```
syn Type Exp.type();             // (1)
eq  Use.type() = decl().type(); // (2)
```

The attributes are evaluated on demand. For example, when the `type` is requested for an instance of `Use`, equation (2) will be evaluated. This will in turn place a request on the `decl` attribute of that `Use`, followed by a request on the `type` attribute of that declaration. Attributes are cached to avoid repeated evaluation of the same attribute instance.

Current systems supporting reference attributes include JastAdd [9], Silver [24], and Kiama [20]. JastAdd has been used for the implementation of JModelica.org, one of the open-source Modelica systems of interest to this paper [1].

The JastAdd specification language is tightly coupled to Java, making use of the Java class hierarchy to implement the syntax tree, and allowing Java code to be used in the equations. In addition to reference attributes, JastAdd supports several other extensions to AGs. Of particular importance to this paper are *higher-order attributes* [22], which allow attributes to themselves be attributed syntax trees. Also of importance is the aspect-oriented specification language of JastAdd that combines object-orientation, inter-type declarations, and attribute grammars, a combination which allows a high degree of modularization [3].

3 Compiling Modelica Models

In this section, we first give an overview of the Modelica compilation process. This is followed by a description of compilation problems and quality aspects that are especially challenging when compiling Modelica models.

3.1 Compilation Process

A typical Modelica compilation process is depicted in Figure 1. Input to the process is a Modelica source code model. The first phase performs traditional parsing and optional simplifications, resulting in an *abstract syntax tree (AST)* representing the model.

The second phase (highlighted in Figure 1) is called the *elaboration* process[4], and is the main focus of this paper. Here, the main model class is instantiated, recursively instantiating all components inside it, resulting in an *instance hierarchy*. For example, elaborating the `Newton` model of Section 2 gives an instance of `Newton` that contains four instances of `Real`. The result of the elaboration phase, the *hybrid DAE*, is achieved by a straightforward traversal of the instance hierarchy, collecting variables and equations.

The hybrid DAE can then be further manipulated using symbolic algorithms in order to reduce its complexity and enable numerical simulation. In a final step the model equations are generated in a language suitable for efficient computation, e.g. C, and then typically compiled and linked with a numerical simulation algorithm.

3.2 Compilation Problems

The elaboration phase in a Modelica compiler is highly challenging. Key tasks include *name analysis*, *type analysis*, and building the *instance hierarchy*. Be-

[4]This process is sometimes in the literature also referred to as the "flattening" or the "instantiation" process.

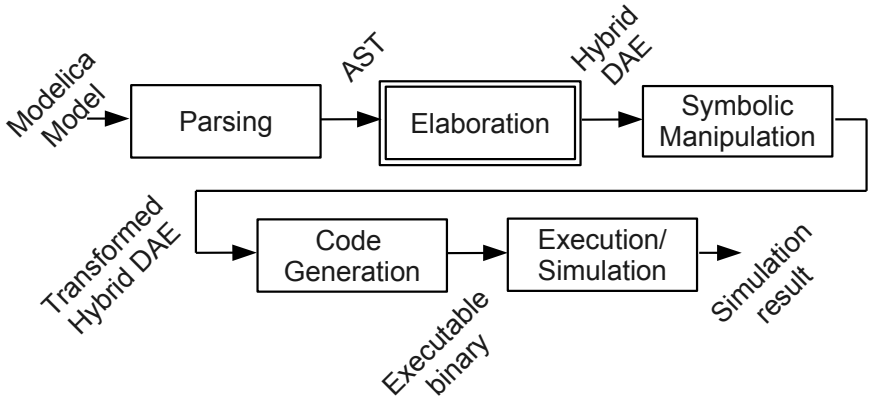

Figure 1: The Modelica compilation phases.

sides being complex, these tasks are also mutually dependent: The compiler will need to alternate between them.

In *name analysis*, the task is to resolve connections between uses and declarations of names, according to the scope rules. In addition to lexical scoping, field selection, and inheritance, as in mainstream programming languages, Modelica names can also be scoped along the instance hierarchy. Furthermore, building the instance hierarchy depends on the name analysis, so these two tasks are mutually dependent.

Modelica has a primarily *structural type system* (as in many functional languages), in contrast to the *nominal* type systems of mainstream object-oriented languages. I.e., type equivalence and subtyping is decided based on the structure of the instances, rather than on class names and inheritance hierarchy [4]. The types influence the scope rules via field selection and inheritance, making all three tasks mutually dependent on each other.

One particularly complex language construct is that of *redeclarations* which has similarities to generics in object-oriented languages. It allows construction of general parameterized (through several levels) models, which is handled by symbolic replacement of parts of classes during compilation. Deep redeclaration hierarchies are challenging to handle efficiently, since they render lookup of names to be context sensitive, and dependent on the instance hierarchy, and an environment of applicable redeclarations needs to be maintained throughout the compilation process.

3.3 Quality Aspects

There are many quality factors to consider when engineering a compiler for a complex language. For a Modelica compiler, three important quality factors are *modularity*, *extensibility*, and *performance*.

By *modularity* we mean that the compiler can be divided into separate independent parts that can be combined together to form a working compiler. We say that it is *highly coupled* if the different modules are strongly dependent on each other. Modularity and low coupling are especially challenging to achieve for the elaboration phase of the Modelica compiler. One modularity conflict is between the desire for early type checking (for good error reporting) and flexible parameterized modeling using redeclarations. The latter is often using partial (incomplete) models making it necessary to postpone checking because all information is not available until the end of the compilation process.

Another important quality factor is *extensibility*. As the Modelica language grows in popularity, uses other than simulation are gaining increased interest. Such uses include optimization, model calibration, model reduction, and uncertainty modeling. Another aspect of extensibility is the problem of maintaining different compiler versions corresponding to different versions of the Modelica language.

Performance is also an important quality factor of Modelica compilers because a common user scenario is to interactively update models and test them in simulation. As a consequence, users go through the cycle *i)* model editing, *ii)* model compilation, *iii)* model simulation and *iv)* simulation result analysis repeatedly. Therefore, the model compilation time affects the overall design cycle time.

In relation to the use of a high-level formalism for implementing the compiler, additional important quality factors are *correctness* of the generated compiler, *ease of specification*, and a low *learning curve* of the formalism.

4 Declarative solutions

In this section, we analyze how the two Modelica compilers OpenModelica and JModelica.org solve the different compilation and quality problems identified in Section 3.

4.1 OpenModelica

OpenModelica [7] is an open source modeling and simulation environment for the Modelica language. The central part of this environment is the OpenModelica Compiler (OMC) that translates Modelica models to executable simulation binaries. OMC is implemented in the RML [16] specification language and compiled to C using the RML tool. It was originally designed as an executable big-step operational semantics specification of the Modelica language [11], but has gradually evolved into an industrially used Modelica compiler.

The compilation steps in OMC are organized similarly as depicted in Figure 1. The source code is divided into a number of *modules*, each consisting of a set of rule sets. For example

```
rule ceval(env,e1) => Values.BOOL(true) &&
     ceval(env,e2) => v
     ------------------------------------
     ceval(env,DAE.IFEXP(e1, e2, e3)) => v
```

is the rule for the `true`-branch for constant evaluation (identifier `ceval`) of expressions. Note the direct correspondence to the BSOS rule given in (1) in Section 2.2.

In order to perform the elaboration process efficiently, the compiler finds out which models (classes) are actually used in the particular model that is being instantiated. This is currently done in a *dependency analysis* pre-phase, including simplified elaboration and name lookup to find out what the elaborated classes actually contain.

The first versions of the OpenModelica compiler did not have a separate dependency analysis phase, causing many models to be unnecessarily elaborated (i.e., performance problems), and sometimes incorrectly elaborated due to tricky cases of mutual dependencies. The separate dependency analysis phase improves to a certain degree both modularity, performance, and correctness.

Name lookup is done in environments which are built on the fly during the analysis. Type representations (as simplified AST trees) are also built on the fly and stored in the environments. Since the Modelica type system is mostly structural, the type representations are traversed recursively in order to determine possible type equivalence or subtyping relationships.

As an example of name lookup, we show part of the rule set called `lookupScope`, which looks up a simple identifier in the environment[5].

[5]The example is a RML translated version of the current OpenModelica implementation where several identifiers are abbreviated.

```
rule avlTreeGet(cls_and_vars,name) => i1 &&
     resolveAlias(i1,cls_and_vars) => i2
     -------------------------------------
     lookupS(_,FRAME(cls_and_vars)::_) =>
     (SOME(i2),SOME(IDENT(name)),SOME(env))
...

rule lookupQImports(inName, imps, env)
       => (opt_item, opt_path, opt_env)
     -------------------------------------
     lookupS(_,FRAME(ITBL(false,imps))::_)
       => (opt_item, opt_path, opt_env)
```

These rules show an example where backtracking is used: if the pattern does not match or a premise fails for a rule, the next rule is tried.

Redeclarations are handled by changing bindings of elements declared in models. This is done by updating environments and building new environments – the original model AST is not modified. To correctly elaborate a model, the whole chain of redeclarations for a model needs to be known. In some tricky cases both the part being replaced as well as the part it is replaced with need to be elaborated before the actual redeclaration can be done. Finally, the equations (i.e., the DAE) are extracted by traversing the model together with its environments.

Extending the compiler is done in three steps: i) extend the AST with new nodes ii) add new RML rules in existing rule sets, and iii) add new rule sets (if applicable). For example, OMC has been extended with partial differential equations (PDEs) and MetaModelica [19]. Extending the AST may affect many rule sets, but the static type system of RML makes sure that no rule extensions are accidentally missed.

To gain good performance of the elaboration phase is a challenging task, both due to the definition of the Modelica language and due to the fact that OpenModelica was originally designed as a specification and not as an efficient implementation.

During recent years, several solutions have been deployed to improve the compiler's performance. More efficient pure algorithms and data structures have been added, e.g. to use AVL trees instead of lists for name lookup. Also a cache mechanism has been introduced for eliminating repeated elaboration of equivalent terms. These pragmatic performance improvements caused the introduction of new (impure) updatable arrays into the RML language.

4.2 JModelica.org

JModelica.org [1], is an open-source platform for simulation and optimization of physical systems with a scope similar to OpenModelica. The Modelica compiler which is part of JModelica.org is implemented in the JastAdd system [9] supporting reference attribute grammars (RAGs) [8].

Name analysis in JModelica.org uses the typical RAG approach that was developed for the JastAddJ Java compiler [6]: `Use` nodes have reference attributes pointing to the appropriate `Decl` nodes in the AST, and these attributes are defined declaratively, making use of parameterized attributes that look up names according to the scope rules, delegating partial computations to other attributes. Thus, no explicit symbol table is used, but the AST itself is used as the symbol table.

The synthesized attribute `lookupDefault` is a typical such attribute from JModelica.org. It looks for the declaration among members and imports, and if not found there, delegates to its attribute `lookup`. `Lookup` is an inherited attribute defined in an enclosing node, and used for handling lexical scope:[6]

```
syn Decl FullDecl.lookupDefault(String n) {
    Decl res = lookupMembers(n);
    if (res.isUnknown())
      res = lookupInImports(n);
    return
      res.isUnknown() ? lookup(n) : res;
}
```

For representing types, reference attributes are used, pointing to appropriate declarations. To compare types, the declaration structure is traversed recursively, using parameterized attributes.

A major difference from Java compilation, however, is that for JModelica.org, the AST traversed during name and type analysis is not the source AST, but an AST representing the instance hierarchy. This *instance AST* is built declaratively using higher-order attributes [22]. Each instance is defined as a higher-order attribute, and contains one higher-order attribute for each of its subinstances. This way, the instance AST can be built gradually, top down. It turns out that this provides a solution to the mutual dependencies between name analysis, type analysis and building the instance hierarchy: The tasks are carried out in an interleaved fashion, automatically scheduled according to the attribute dependencies by the RAG evaluation machinery [2]. Here is a very simplified part of the instance AST specification:[7]

```
Inst ::= /Inst*/ ...;
eq Inst.getInstList() =
   ... new Inst(...) ...;
```

The approach fits well also with the redeclaration construct which can modify types and variables for individual instances, and thereby influence both individual instances and the structure of the instance hierarchy. To handle them in JModelica.org, a modification environment is built using a higher-order attribute for each instance, as a result of merging environments in enclosing instances with local modifications [2]. The computation of these environments is automatically interleaved with the other analyses by the RAG evaluator.

The JModelica.org compiler is specifically designed to be modular and extensible, relying on declarative RAG specification and static aspects as supported by JastAdd. Individual problems like name analysis, type analysis, redeclarations, and instance building are defined as separate modules with low coupling, although the tasks become closely intertwined when running the compiler. The compiler can be extended simply by adding more modules that define more abstract syntax and/or new attributes. As an example, an extension language *Optimica* has been defined, targeting dynamic optimization of Modelica models. Optimica has been implemented as a completely modular extension to JModelica.org [1].

[6] Identifiers are abbreviated. The equation is given directly in the form of a method body. Assignments are allowed as long as there are no side effects outside the body.

[7] A right-hand side nonterminal within slashes is at the same time an attribute. Rather than the parser building this part of the AST, it is defined by an equation.

The JModelica.org compiler is between 5 and 10 times slower than commercial Modelica tools, but this is still sufficient for solving many problems. Examples of commercial projects using JModelica.org include the development of control systems for post combustion CO_2 separation systems, and the optimization of models for polyethylene production.

5 Concluding Discussion

Both OpenModelica and JModelica.org demonstrate how formal languages can be used for specifying complex languages and for generating mature compilers used by industry. The OpenModelica system has been developed since the first release of the Modelica specification and is a more mature system, supporting almost the full Modelica standard. JModelica.org is newer, not supporting full Modelica yet, but with a focus on language extensions, in particular for model optimization.

In the following discussion, we compare the implementation of Modelica based on the two approaches: i) big-step operational semantics (implementation of OpenModelica using RML) and ii) reference attributes grammars (implementation of JModelica.org using JastAdd). We use the abbreviations BSOS and RAGs to denote the whole implementation chain BSOS/RML/OpenModelica and RAGs/JastAdd/JModelica.org respectively.

In trying to compare the approaches of BSOS and RAGs, it is clear that the focus and goals during the implementation have been slightly different for the two systems. It is therefore not really possible to say that one approach is better than the other, or that one issue solved particularly well in one approach could not have been done well also in the other approach. Instead, we will try to pinpoint key issues that are solved in similar or differing ways, and key advantages that either of the two approaches have shown, as compared to using ordinary programming languages for implementation.

5.1 Approaches to compilation problems

It is interesting that while both BSOS and RAGs are declarative, the solutions to compilation problems are quite different. For name analysis, the BSOS approach is building environments with declaration information and passing them as parameters between rules. A RAG, in contrast, uses the AST itself to represent bindings between declarations and identifiers, making use of reference attributes to connect different parts of the AST into a graph.

The type analysis is fairly similar in both approaches, although done on different data structures: both use recursion to analyze the two types to be compared.

Building the instance hierarchy is done using very different approaches. For the BSOS approach, the computation is performed in several explicit phases, where the final instantiation phase recursively calls instantiation and name lookup. In contrast, in the RAGs approach the phases are implicit and scheduled automatically by the attribute evaluator.

5.2 Support for quality aspects

Both the BSOS and the RAGs approach are high-level rule-based declarative specification languages that abstract away from certain scheduling of computations: rule selection and backtracking for BSOS, and attribute evaluation order for RAGs. In the RAGs case, these properties are especially important for *ease of implementation*, as was particularly apparent for implementing the dependent name, type, and instance analyses in JModelica.org. In the BSOS case, the mechanisms of rule selection and backtracking are vital for the correspondence to the rule based operational formalism, making it fairly easy to efficiently compile BSOS specification.

The declarative approach also lends itself to *modularization* and *extensibility*, as was demonstrated by the extension of OpenModelica with PDEs and MetaModelica, and by the extension of JModelica.org with the Optimica language. However, where extensibility has been one of the primary goals with JastAdd, it is less emphasized as an objective for RML.

Concerning the *correctness* of the generated compilers the primary aids in the BSOS approach are both the static type checking of rules in RML and the direct correspondence to the specification formalisms of big-step operational semantics. For example, the first version of the RML/OpenModelica specification was one of the first steps of defining the semantics of Modelica [11]. In the RAGs case, high level of correctness is achieved by the possibility to extend the language by adding rules, rather than changing an existing implementation. This makes it easier to ensure that existing parts of the language are unaffected.

Concerning *performance*, both OpenModelica and JModelica.org are currently slower than commercial tools. But not dramatically so, and they have proven themselves sufficiently fast for practical use on many industrial problems. By improving the metacompilation tools (RML and JastAdd), performance improvements can be applied to all languages implemented using these tools.

Finally, there will naturally be a *learning curve* to use a formal specification language as opposed to a general-purpose programming language. Here RML and JastAdd use very different approaches. RML is modelled closely on BSOS and is well suited for use by academics with training in formal semantics and functional programming. JastAdd is instead very close to Java in its syntax and its object-oriented model of the AST, and is fairly easily understood by programmers in industry. To lower the BSOS learning curve for Modelica programmers, MetaModelica was recently developed, a new RML-like language but with Modelica-like syntax [19].

5.3 Future work

Our analysis has been exploratory and qualitative, and could serve as a starting point for a quantitative evaluation. In particular, it would be interesting to define a tiny language that exhibits the mutual dependencies between name analysis and building the instance hierarchy, and that could serve as a benchmarking problem for different metacompilation approaches.

References

[1] J. Åkesson, K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, Nov. 2010.

[2] J. Åkesson, T. Ekman, and G. Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1-2):21–38, Jan. 2010.

[3] P. Avgustinov, T. Ekman, and J. Tibble. Modularity first: a case for mixing AOP and attribute grammars. In *7th Int. Conf. on Aspect-Oriented Software Development (AOSD'08)*, pages 25–35. ACM, 2008.

[4] D. Broman, P. Fritzson, and S. Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, pages 303–315, Vienna, Austria, 2006.

[5] T. Despeyroux. TYPOL - A Formalism to Implement Natural Semantics. Rapports Techniques no 94, INRIA - Sophia-Antipolis, 1988.

[6] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.

[7] P. Fritzson, P. Aronsson, H. Lundvall, K. Nystrm, A. Pop, L. Saldamli, and D. Broman. The OpenModelica Modeling, Simulation, and Software Development Environment. *Simulation News Europe*, 15(44/45):8–16, 2005.

[8] G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[9] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[10] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196, 1996.

[11] D. Kågedal and P. Fritzson. Generating a Modelica compiler from natural semantics specifications. In *Summer Computer Simulation Conference (SCSC'98)*, pages 299–307, Reno, Nevada, 1998.

[12] G. Kahn. Natural semantics. In *4th Annupal Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, volume 247 of *LNCS*, pages 22–39, Passau, Germany, 1987. Springer.

[13] D. E. Knuth. Semantics of Context-free Languages. *Math. Sys. Theory*, 2(2):127–145, 1968. Correction: *Math. Sys. Theory* 5(1):95–96, 1971.

[14] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[15] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[16] M. Pettersson. *Compiling Natural Semantics*, volume 1549 of *LNCS*. Springer, 1999.

[17] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[18] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Department of Computer Science, University of Aarhus, 1981.

[19] A. Pop and P. Fritzson. Metamodelica: A unified equation-based semantical and mathematical modeling language. In *Modular Programming Languages (JMLC'06)*, volume 4228 of *LNCS*, pages 211–229, Oxford, 2006. Springer.

[20] A. M. Sloane, L. C. L. Kats, and E. Visser. A pure object-oriented embedding of attribute grammars. *Electr. Notes Theor. Comput. Sci.*, 253(7):205–219, 2010.

[21] A. Verdejo and N. Marti-Oliet. Executable structural operational semantics in Maude. *The Journal of Logic and Algebraic Programming*, 67(1–2):226–293, 2006.

[22] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.

[23] D. Wile. Supporting the DSL spectrum. *J. of Comp. and Inf. Tech.*, 9(4):263–287, 2001.

[24] E. V. Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: An extensible attribute grammar system. *Science of Computer Programming*, 75(1-2):39–54, 2010.