A comparison of two metacompilation approaches to implementing a complex domain-specific language

David Broman
Linköping University, Sweden
david.broman@liu.se

Peter Fritzson
Linköping University, Sweden
peter.fritzson@liu.se

Görel Hedin
Lund University, Sweden
gorel.hedin@cs.lth.se

Johan Åkesson
Lund University, Sweden
johan.akesson@control.lth.se

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.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compiler generators*; I.6.2 [**Simulation and Modeling**]: Simulation Languages

Keywords

metacompilation, operational semantics, attribute grammars

1. INTRODUCTION

Computer languages constitute a key way of supporting problem solving in different domains. The use of *metacompilers*, i.e., languages for the compiler domain, allows compilers and related tools to be generated from high-level specifications, bringing down down the otherwise very high tool development costs [8]. Such metacompilers can be based on different formalisms such as operational semantics, denotational semantics, attribute grammars, and algebraic term rewriting.

The goal of this paper is to provide insight into how such approaches can be applied to complex domain-specific languages in industrial use. In particular, how are key compilation problems solved in the different approaches?

As a case study, we study compilers for the language *Modelica*[1], a language for modeling and simulation of physical systems, based on hybrid differential-algebraic equations (DAEs). There is a number of compilers for Modelica, two of which are implemented using metacompilation:

OpenModelica[2] which is implemented using the RML metacompiler [9], based on big-step operational semantics (BSOS) [7]; and *JModelica.org*[3] which is implemented using the JastAdd metacompiler [5], based on reference attribute grammars (RAGs) [4]. 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.

This paper is based on the report [3].

2. IMPLEMENTING MODELICA

The Modelica language allows users to express physical relations of model components using mathematical differential and algebraic equations, and components can be connected, acausally, to create complex composite models. Modeling of discrete behavior is also supported, allowing, e.g., modeling of control systems. Object-oriented constructs like classes, inheritance and generics, can be used to specify reusable component types. Class libraries support different modeling domains like electronics, mechanics, and thermodynamics.

In the following 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, and force, using time-derivatives (the `der` function).

The model can be used to simulate the behavior of variables like `pos` and `vel` as a function of time.

```
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;
```

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

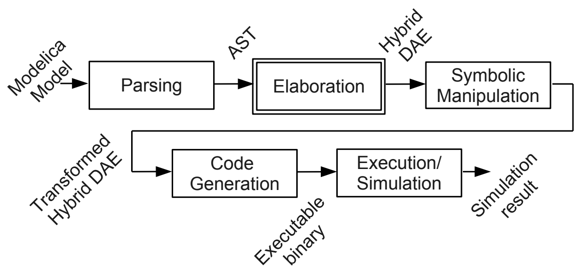

Figure 1: The Modelica compilation phases.

Typically, models consist of many objects, of different model classes, connected together. A complete model may contain tens of thousands of individual equations.

A typical Modelica compilation process is depicted in Figure 1. The main focus of this paper is the *elaboration* phase, also called "flattening" or "instantiation". Here, the main model class is instantiated, followed by recursively instantiating all components inside it, resulting in an *instance hierarchy*. For example, elaborating the `Newton` model 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 elaboration phase in a Modelica compiler is challenging. It includes the key tasks of *name analysis*, *type analysis*, and building the *instance hierarchy*. These tasks are all mutually dependent, and the compiler will need to alternate between them. For instance, Modelica names can be scoped along the instance hierarchy, and not only along the lexical and inheritance hierarchies. The types influence the scope rules via field selection and inheritance. Further, the type system is mostly structural, and type equivalence and subtyping is decided based on the structure of the instances, rather than on class names and inheritance hierarchy [2].

One particularly complex language construct is that of *redeclarations* which has similarities to generics in object-oriented languages. It allows the construction of general models that are parameterized through several levels, which is handled by symbolic replacement of parts of classes during compilation.

3. BIG-STEP OPERATIONAL SEMANTICS

The OpenModelica compiler (OMC) is implemented using RML [9], a metacompiler based on Kahn's big-step operational semantics (BSOS), originally called natural semantics [7]. Rules are defined as axioms or inference rules and are grouped into collections. For performance reasons, RML rules are ordered, and evaluated in a more functional style than the unordered relational rules in BSOS. However, like in BSOS, and in contrast to functional languages, RML includes backtracking, i.e., if a premise fails, backtracking of the rule occurs and the next rule in the collection is executed. OMC was originally designed as an executable specification of Modelica [6], but has gradually evolved into an efficient implementation.

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. The rule can be read clock-wise as follows: If expression `e1` in environment `env` is evaluated to `true` then evaluate `e2` 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*.

In order to perform the elaboration process efficiently, OMC includes a dependency analysis pre-phase, which finds out which models (classes) that are actually used in the particular model that is being instantiated, thereby avoiding to unnecessarily elaborate other models.

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 an abbreviated part of the rule set called `lookup`, which looks up a simple identifier in the environment

```
rule avlTreeGet(cls_and_vars,name) => i1 &&
     resolveAlias(i1,cls_and_vars) => i2
     ------------------------------------
     lookup(_,FRAME(cls_and_vars)::_) =>
     (SOME(i2),SOME(IDENT(name)),SOME(env))
...
rule lookupQImports(inName, imps, env)
       => (opt_item, opt_path, opt_env)
     ------------------------------------
     lookup(_,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.

4. REFERENCE ATTRIBUTE GRAMMARS

The JModelica.org compiler (JMC) is implemented using JastAdd [5], a metacompiler based on reference attribute grammars (RAGs) [4]. In attribute grammars, 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.

RAGs extend attribute grammars by allowing attributes to be references to syntax nodes, and to be dereferenced to access non-local attributes. 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)
```

In JastAdd, 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.

To define the value of the decl attribute, JMC uses *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 in JMC. 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:[4]

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

The AST traversed during name and type analysis is not the source AST, as for normal programming languages, but an AST representing the instance hierarchy. This *instance AST* is built declaratively using another extension to AGs called *higher-order attributes* (HOA) [10]. A HOA is an attribute that is itself an AST, and that can have its own attributes. Each model instance is defined as a HOA, and contains one HOA 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 [1]. Here is a very simplified part of the instance AST specification (a right-hand side nonterminal within slashes is a HOA):

```
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 the structure of the instance hierarchy. To handle this in JMC, a modification environment is built using a HOA for each instance, as a result of merging environments in enclosing instances with local modifications. The computation of these environments is automatically interleaved with the other analyses by the RAG evaluator.

[4]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.

5. CONCLUDING DISCUSSION

Both OMC and JMC demonstrate how formalisms like BSOS and RAGs can be used for specifying complex languages and for generating mature compilers used by industry.

While BSOS and RAGs are both declarative formalisms, some compilation problems are solved in quite different ways. JastAdd RAGs focus on extensibility whereas the BSOS-based RML focuses on conformance with formal specifications.

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.

6. REFERENCES

[1] 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.

[2] 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.

[3] D. Broman, P. Fritzson, G. Hedin, and J. Åkesson. A comparison of metacompilation approaches to implementing Modelica. Technical Report 1404-1200 Report 97, Dept. of Computer Science, Lund University, Sweden, 2011.

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

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

[6] 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.

[7] 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.

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

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

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