Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 203 (2008) 117–131

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Development of a Modelica Compiler Using JastAdd

Johan Åkesson[1]

Department of Automatic Control, Lund University, Lund, Sweden

Torbjörn Ekman[2]

Computing Laboratory, Oxford University, Oxford, United Kingdom

Görel Hedin[3]

Department of Computer Science, Lund University, Lund, Sweden

Abstract

This paper describes experiences from implementing key parts of a compiler for Modelica, an object-oriented language supporting declarative modeling and simulation of complex physical systems. Our implementation uses the attribute-grammar based tool JastAdd. In particular, we discuss the implementation of Modelica *name analysis* which is highly context-dependent, *type analysis* which is based on structural subtyping, and *flattening* which is a fundamental part of the Modelica compilation process.of so called *modifications*, Modelica.

Keywords: Modelica, JastAdd, compiler construction, reference attributed grammars

1 Introduction

The Modelica language is widely used in industry for modeling and simulation of physical systems [17]. Example application areas include industrial robotics, automotive applications and power plants. Typical for models of such systems is that they are of large scale, commonly up to 100.000 equations. There are also extensive standard libraries for e.g., mechanical, electrical, and thermal models.

In the area of modeling of physical systems, most software available today is focused on supporting *simulation*. That is, prediction of the response of a model,

[1] Email: jakesson@control.lth.se

[2] Email: torbjorn@comlab.ox.ac.uk

[3] Email: gorel@cs.lth.se

doi:10.1016/j.entcs.2008.03.048

given a specified input stimuli. If the model is sufficiently accurate, the simulation results may be used to draw conclusions about the behavior of the true physical system. The term *model* is here used to refer to a mathematical description of the behavior of a physical system. For example, a simple model of a car would implement Newton's second law, which relates acceleration to applied force.

While current software is very efficient when simulating models, it does often not offer the flexibility needed to use models for other purposes. One example is dynamic optimization. In the car example above, it might be interesting to find a force trajectory, which minimizes the time it takes to transfer the car from one point to another. Apart from simulation and optimization, there are also many other interesting uses of large scale models. These emerging fields have in common that they offer numerous methods and algorithms, suitable for different model structures.

Currently, it is difficult, if not impossible, to apply this wide range of available and emerging methods on models developed in Modelica. In particular, these new fields require new high level descriptions, which complement or extend current modeling languages. It is highly desirable that such extensions are done in a modular way, so that the actual *model* description is separated from, potentially several, *complementing* descriptions, relating to the model.

As a fist step towards creating a flexible Modelica-based modeling environment that can support these emerging methods, a new extensible compiler, entitled JModelica, is under development. For this development we use the compiler construction tool JastAdd [11,7]. This tool uses several declarative features such as reference attribute grammars and rewriting, in order to support building extensible compilers.

In this paper we describe our experience from building a prototype of the JModelica compiler, and how the complex compilation problems in Modelica can be solved in JastAdd in a modular, understandable and compact manner. In addition to name and type analysis, where Modelica has advanced context-dependent rules, we discuss *flattening* of Modelica models. Flattening is the process of eliminating hierarchical constructs and producing a flat model description which can be, after some further processing, interfaced with numerical algorithms.

The paper is organized as follows. In Section 2, the compiler tool JastAdd is described. Section 3 gives an overview of the Modelica language. The paper proceeds with a description of how key features in the Modelica language have been implemented using JastAdd in Section 4. In Section 5 the performance of the JModelica compiler is compared to two other Modelica tools. The paper ends with Section 6 on conclusions and future directions.

2 JastAdd

The JastAdd compiler construction system combines a number of features including object-orientation, intertype declarations, reference attribute grammars, circular attributes, and context-dependent rewrites. The goal is to allow high-level executable specifications that support building extensible compilers. For example, the system has been used for building a complete Java 1.4 compiler and modular extensions to support Java 5 [7]. Current state-of-the art hand-coded compilers do not support the fine-grained extensibility that is needed for such extensions, but usually have support only for simple modularization into separate compiler phases, e.g., using the Visitor design pattern. JastAdd is implemented in Java, generates Java code, and is available under an open-source license [12]. This section gives a brief summary over the main features of importance for this paper.

Object oriented abstract syntax The core of a language implementation is specified by an abstract grammar. JastAdd generates a Java class hierarchy from the grammar, including constructors and traversal API. For example, a grammar rule `WhileStmt:Stmt ::= Cond:Exp Do:Stmt;` will generate a class `WhileStmt` which extends `Stmt`, with a constructor `WhileStmt(Exp e, Stmt s)`, and two traversal methods `Exp getCond()` and `Stmt getDo()`. Any Java-based parser generator can be used for building the abstract syntax tree (AST), using the constructors.

Aspects with intertype declarations Behavior of the AST classes is added using aspect modules that support intertype declarations. For example, a feature `f` that belongs to a class `A`, can be declared in an aspect module as `A.f`. The syntax is similar to the intertype declarations in AspectJ [13], but dynamic aspect-oriented features like pointcuts etc., are not supported. Typically, each different compiler problem, like name analysis, type analysis, code generation, etc., is defined in one or several aspect modules.

Reference attribute grammars Behavior is primarily defined through attributes, whose values are defined by equations. An equation defines the value of one attribute in terms of the values of other attributes in the AST. Like in ordinary Knuth-style attribute grammars [14], *synthesized* and *inherited* attributes are used for propagating data upwards and downwards in the AST. In contrast to Knuth-style AGs, attributes can be *reference-valued*. This means that an attribute may be a reference to another node, arbitrarily far away in the AST, and other data (attributes) can be accessed via the reference attribute [10]. Typically, this is used for representing name bindings, e.g., from variable use to declaration, from class to superclass, etc. The attribute declarations and equations are specified using intertype declarations in aspect modules, using the keywords `syn`, `inh`, and `eq`.

Parametrized attributes An equation for an attribute is defined as a member of a class, and can be seen as having `this` as an implicit parameter, giving access to all other attributes in the same node (and possibly others via reference attributes or AST traversal). JastAdd supports also explicitly parametrized attributes, giving the equation access to more, possibly non-local, information. Typical uses include name lookup and type comparisons. Examples will be given in Section 4.1.

Additional declarative features JastAdd supports conditional **rewrites** that can use attributes to define context-dependent modifications to the AST. Rewrites are typically used for modifying the AST from the initial form constructed by the parser, to a form more suitable for compilation [5]. JastAdd also supports **circular attributes** that are evaluated using fix-point iteration [8,16], and **non-terminal attributes** [21] that are AST nodes defined by equations. To avoid

```
class BouncingBall //A model of a bouncing ball
  parameter Real g = 9.81; //Acceleration due to gravity
  parameter Real e = 0.9; //Elasticity coefficient
  Real pos(start=1); //Position of the ball
  Real vel(start=0); //Velocity of the ball
equation
  der(pos) = vel;   // Newtons second law
  der(vel) = -g;
  when pos <=0 then
    reinit(vel,-e*pre(vel)); // set velocity after bounce
  end when;
end BouncingBall;
```

Figure 1. A class modeling bouncing balls

the copy rules that clutter a canonical attribute grammar, a mechanism called **broadcasting** is used, similar to the *including* mechanism [15].

Combining declarative and imperative code While most of the compilation is best defined by the declarative attributes, there will usually be a need for generating some output based on the attributed AST. The attributes constitute an API that can be used by imperative code, i.e., ordinary Java methods. In the JModelica compiler, the flat model is constructed imperatively, using the attribute API defined by the declarative name and type analysis.

3 Modelica

The Modelica language has evolved from the simulation community, with roots in analog simulation dating back to the 1940's. For an overview of the evolution of the field of continuous time modelling and simulation, see [1].

The first version of Modelica was published in September 1997. The effort was targeted at creating a new general-purpose modeling language, applicable to a wide range of application domains. While several other modeling languages were available, many of those were domain-specific, which made simulation of complex heterogeneous systems difficult. Based on experiences from designing other modeling languages, notably Omola, [2], the fundamental concepts of *object-orientation* and *declarative programming* were adopted. The latest version of the Modelica specification, 2.2, (see [20]) was released in February 2005.

3.1 An introductory example

Modelica is an object-oriented language in that it uses objects, classes, and inheritance to define and specialize models of physical entities. A fundamental difference from an ordinary programming language, like Java, is the way behavior is defined: In Modelica, there is no concept of run-time state, dynamic object allocation, or method dispatching. What a Modelica program does is to define a number of statically allocated objects, called *components* in Modelica terminology. The behavior of the individual components is defined using equations that typically capture laws of nature. Classes and multiple inheritance are used for abstracting and specializing the components and their behavior.

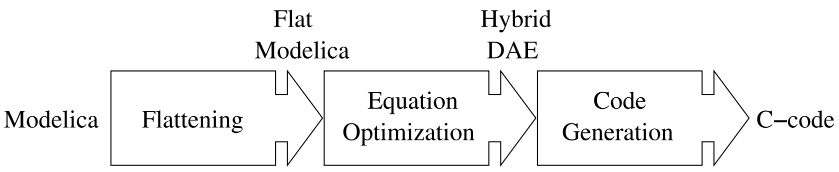


Figure 2. The Modelica translation process.

For example, a class BouncingBall can be defined to model the behavior of bouncing balls, where Newton's second law can be described using equations, see Figure 1. The example illustrates several key features of Modelica:

Classes and components The description of the bouncing ball is encapsulated in a class which consists of a set of component declarations and a set of equations. In this case all components are of type `Real`, but it is also possible to build aggregate object structures where components are instances of user defined classes.

Modification The built-in type `Real` is associated with a set of attributes, such as start value and unit. In the declaration of `pos`, e.g., the start value is set to 1. This mechanism is called *modification*. In more elaborate cases, modifications can be used to express generics, by applying *redeclaration* modifications to local classes or components, when declaring a component. In this way, elements of a class can be parametrized and may be replaced when declaring a component.

Equations and derivatives The behavior of the class *BouncingBall* is defined by two equations, which relate its variables. The function `der` represents the time derivative operator $\frac{d}{dt}$

Instantaneous events The `when`-clause expresses an instantaneous event, i.e., *when* a specified condition evaluates to true, some actions should be taken, e.g., to re-initialize state variables. In the BouncingBall example, the bounce is detected, and the new velocity in the opposite direction is set, taking the elasticity coefficient into account.

We have no space here to cover all important features of Modelica. A thorough and comprehensive description of Modelica and its usage is given in [9].

3.2 Modelica compilers

The objective of the Modelica compilation process is to output code (usually C-code) to be compiled and linked with a numerical simulation package. This process can be divided into a number of steps, see Figure 2. In the first step, the Modelica code is *flattened*. This means that all component, class, and inheritance structures are eliminated. The resulting model contains, essentially, a set of variables and a set of equations. The only property of the flat model that indicates its hierarchical origin is that a variable name is usually expressed as a qualified name, indicating the path of the corresponding variable. Consider the following model which contains two components of the `BouncingBall` class in Figure 1:

```
class BBex
  BouncingBall eBall;
```

```
fclass BBex
  parameter Real eBall.g = 9.81;
  parameter Real eBall.e = 0.9;
  parameter Real mBall.g = 1.62;
  parameter Real mBall.e = 0.9;
  Real eBall.pos(start = 1);
  Real eBall.vel(start = 0);
  Real mBall.pos(start = 1);
  Real mBall.vel(start = 0);
equation
  der(eBall.pos) = eBall.vel;
  der(eBall.vel) =  -eBall.g;
  when eBall.pos <= 0 then
    reinit(eBall.vel,
    -eBall.e*pre(eBall.vel));
  end when;
  der(mBall.pos) = mBall.vel;
  der(mBall.vel) =  -mBall.g;
  when mBall.pos <= 0 then
    reinit(mBall.vel,
    -mBall.e*pre(mBall.vel));
  end when;
end BBex;
```

Figure 3. A flat Modelica description.

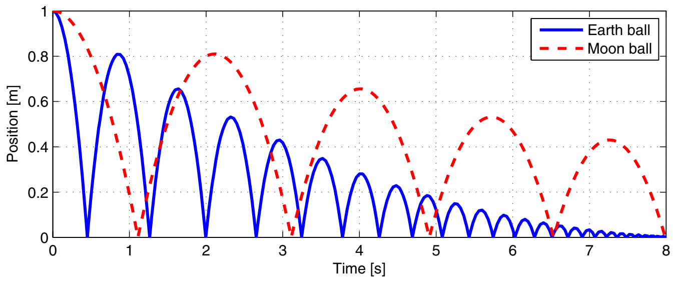


Figure 4. Position as a function of time for the `BouncingBall` example.

```
  BouncingBall mBall(g=1.62);
end BBex;
```

The component `mBall` corresponds to a ball bouncing on the moon, and accordingly, the acceleration due to gravity is changed in a modification. This example also illustrates *merging* of modifications, where outer modifications, in this case `g=1.62`, overrides the inner modifications. The result of the flattening operation is shown in Figure 3. In addition, in Figure 4 the positions of the two balls as a function of time are shown.

In the next step, the equations are sorted, analyzed and optimized so that they can be used with numerical software. The output of this step is referred to as a hybrid differential algebraic equation (DAE), which is a mathematical object. The hybrid DAE also represents a generic mathematical description of the original Modelica model, and may be used for different purposes. The most common application is to generate C code, which is compiled and linked with an algorithm for numerical integration. The behavior of the system can then be simulated by executing the resulting application. The graphs in Figure 4 is an example of a simulation result.

There are several compilers available for Modelica: one commercial product is Dymola from Dynasim, [4] and an open-source alternative is OpenModelica, [18]. Our own prototype compiler, JModelica, currently performs only the flattening step for a subset of Modelica. In order to validate the results, the flat descriptions produced by JModelica and Dymola were (automatically) compared and verified. We compare JModelica to both Dymola and OpenModelica in section 5.

4 Compiler Design

The aim of the Modelica compiler under development, the JModelica compiler, is to support the complete Modelica language. Since Modelica is a large and complex language, a subset of Modelica, entitled PicoModelica (PM) has been defined for the purpose of illustrating the compiler design concepts in this paper. PM contains several of the characteristic features of Modelica, such as classes, components, inheritance, equations and modifications. Also, the design concepts used in the PM compiler are used also in the JModelica compiler. The source code of the PM compiler is available on the PicoModelica website, [19]. In this section we give an overview of how the key problems of name analysis, type analysis and flattening are solved using the JastAdd mechanisms.

4.1 Name analysis

Name analysis in Modelica is challenging because it is highly context sensitive. In addition to the usual mechanisms such as qualified names and inheritance, there are also some very Modelica-specific language features such as *modification*, that affect name analysis.

Let us first consider ordinary qualified names like `A.B.C`. Here, C should be looked up in an environment decided by B, which in turn should be looked up in an environment decided by A. The environments may potentially be complex due to inheritance from other classes. To implement such normal OO name analysis we have applied the same basic design as in the JastAdd Java implementation [6], namely *delegating lookup attributes*.

To implement name lookup, the usual approach in attribute grammars is to use an inherited attribute *env* which contains all visible symbols at that point in the AST. This works fine for simple block-structured scope, but becomes cumbersome when dealing with qualified names and inheritance. In [6] and in our JModelica compiler, the environment is instead defined as an inherited parametrized attribute with the typical signature `ComponentDecl lookupDecl(String)`. Equations implementing lookup delegate to other lookup attributes as needed, often via reference attributes.

When using lookup attributes, the definition of a reference attribute `myDecl` that binds an access of a name to its appropriate declaration becomes a simple equation:

```
eq Access.myDecl() = lookupDecl(getID());
```

Each node that can have an Access child must define the lookupDecl attribute for that Access. Consider a qualified access, modeled by a `Dot` node with the following abstract syntax definition:

```
Dot : Access ::= Left:Access Right:Access;
```

To implement lookup for the right-hand side, the `Dot` node delegates to the class of the left-hand side, as follows

```
eq Dot.getRight().lookupDecl(String name) = getLeft().myClass().memberDecl(name);
```

Here, Dot defines the attribute lookupDecl of its child getRight. The value is found by accessing the reference attribute myClass of the getLeft child and delegating to member- Decl. The parametrized attribute memberDecl is implemented by searching among the declarations in the class, delegating to superclasses if needed. The complete implementation of the name lookup for the OO features of PicoModelica consists of 4 attributes and 9 equations, which are encoded in the aspect module NameLookupForOO.

The Dot node does not need to explicitly define the attribute lookupDecl of its left access, because there is an equation higher up in the AST which gives a default definition of this attribute for the whole subtree, using the attribute broadcast mechanism in JastAdd.

4.1.1 Handling Modelica modifications

We will now look at how the technique of delegating lookup attributes can be extended to handle the Modelica *modifications* that were exemplified in Section 3.

Consider the following component declaration A a(b=c) which declares a to be a component of class A, but modified so that the variable b has the value c (rather than the value defined in the class A). In this case, A and c are looked up in the normal environment, e.g., the enclosing class. However, b should be looked up in the environment defined by class A. To implement this behavior, the following equation is added:

```
eq ComponentModification.getAccess().lookupDecl(String name) =
   lookupDeclInClass(name);
```

which says that the lookupDecl attribute for the getAccess child of a ComponentModification is defined by the attribute lookupDeclInClass(name). This attribute, in turn, is defined as an inherited attribute of the AST class Argument. Since an Argument node may be the child of several different kinds of nodes, all these need to define the lookupDecl- InClass attribute. In particular, to handle the example above, the ComponentDecl node needs to define this attribute for its getModification child, delegating the lookup to the declaration's class:

```
eq ComponentDecl.getModification().lookupDeclInClass(String name)
   = myClass.memberDecl(name);
```

A similar mechanism is the *redeclare* feature. The complete implementation for redeclare and modification in PicoModelica consists of 2 attributes and 6 equations, which are defined in the aspect module NameLookupForModifications.

These examples illustrate how complex semantic rules can be encoded in a compact manner in JastAdd, and how different aspects of the specification can be modularized, i.e., separating the specification that handles *modifications* and *redeclares* from the specification that handles ordinary OO name lookup.

4.2 Type Analysis

Modelica has a structural type system in which relations such as type equivalence and subtype are determined by the structure of the types rather than through explicit declarations. Two unrelated classes may thus be type equivalent if they declare the same named elements, i.e., local classes and components.

4.2.1 Type representation

The concept of types is not formally defined in the Modelica specification, [20]. However, a framework for representing Modelica types is proposed in [3] introducing two different kinds of types, *classtype* and *objtype*, for classes and components, respectively. A classtype is a hierarchical structure with a set of member declarations that are either of classtype or objtype. An objtype is also hierarchical, but its set can only contain declarations of objtypes, and not of classtypes.

In the JModelica compiler we have chosen to represent the types as explicit objects in the AST. This allows us to use the attribute grammar mechanism also for the types and is useful both for defining the set of members of a type, and for defining the subtype relation, as will be illustrated.

The type objects are added using nonterminal attributes. Each `ClassDecl` node will have `ClassType` child, declared as a nonterminal attribute by enclosing it in forward slashes in the abstract grammar:

```
abstract ClassDecl ::= Name:IdDecl /ClassType/;
```

The definition of the `ClassType` nonterminal attribute is specified by declaring it as a synthesized attribute and providing an equation:

```
syn ClassType ClassDecl.getClassType();
eq Model.getClassType() = new ModelType();
```

Here, `Model` is a subclass of `ClassDecl`, and other equations are given for the other subclasses of `ClassDecl`, for example, to handle predefined Modelica classes. Each `Component- Decl` will similarly have an `ObjType` child, defined by a nonterminal attribute in a similar way.

In order for a `ClassType` to get access to its parent `ClassDecl`, an inherited reference attribute `myClass` is defined:

```
inh ClassDecl ModelType.myClass();
eq ClassDecl.getClassType().myClass() = this;
```

The declarations for a `ClassType` is defined as an attribute `structMap`. The method syntax looks imperative, but has no external side effects and is thus declarative:

```
// map from member name to objtype or classtype references
syn HashMap ModelType.structMap() {
  HashMap result = new HashMap();
  for(Iterator iter=myClass().members().iterator();iter.hasNext();)
    // add objtype and classtype references to the result
  return result;
}
```

Similarly, an `ObjType` can be linked back to its `ComponentDecl`, and its `structMap` attribute can be computed. Notice that the `structMap` contains *references* to other `ClassType` or `ObjType` objects, from which other parts of the AST can be accessed.

4.2.2 Subtype computation

The most important operation when typechecking the Modelica language is to compute if two types are in the subtype relation. A `ModelType S` is a subtype of another `ModelType T` if the following two conditions hold:

- For each member `M` in `T` there is a member `N` in `S` with the same name.
- The type of `N` in `S` is a subtype of the type of `M` in `T`.

This kind of covariant subtyping is safe since the components of a class are only assigned values when the class is instantiated, at which time the exact type is known. Before implementing the general subtype relation between two general `Type` objects, we start by implementing the supertype relation between a `Type` and a `ModelType` object by introducing the parametrized attribute `superModelType` below. (`Type` is a superclass to `ModelType`.)

```
syn boolean Type.superModelType(ModelType subType) circular [true];
eq ModelType.superModelType(ModelType subType) {
  for(Iterator i = structMap().keySet().iterator();i.hasNext();) {
    String name = (String)i.next();
    Type member = (Type)subType.structMap().get(name);
    if(member==null||!member.subType((Type)structMap().get(name)))
      return false;
  }
  return true;
}
```

The attribute is declared circular because erroneous programs may contain cyclic inheritance chains which will incur a circular dependency in the attribute. JastAdd supports evaluation of circular attributes through fix-point iteration [8,16]. In such evaluation, termination is guaranteed if the value domain forms a finite height lattice and the function is monotonic. The example satisfies both conditions with a boolean lattice and an equation with bottom value *true* and a meet operation that reaches top for a single *false* element.

The example above showed the supertype computation for `ModelTypes`. When computing the subtype relation it is convenient to have different rules for different kinds of types. Since Java only supports dispatch on the receiver we use the *double dispatch* pattern to implement a binary method where the target method is selected based on the type of both the receiver and the argument. Consider the additional code below for the subtype attribute. This code dispatches on the type of the receiver, e.g., ModelType, to select a second attribute that is dispatched on the argument. Other type combinations can be added in a modular fashion through additional *subtype* and *supertype* equations in `Type`'s subclasses [7].

```
syn boolean Type.subtype(Type t);
eq ModelType.subtype(Type t) = t.superModelType(this);
```

The entire type analysis framework is implemented in the PicoModelica module `Type- Analysis`, using 10 attributes and 12 equations.

4.3 Flattening

A model in Modelica is defined using static composition of components. The entire concept of nested components can therefore be eliminated at compile-time using a technique called flattening. All hierarchical constructs are removed and the result is a flat description consisting of a set of uniquely named variables of primitive type and a set of equations that operate on primitive values and variables only.

We first present a simplified implementation where modifications are not allowed, and then we discuss how it can be extended to take modifications into account.

4.3.1 Simplified flattening without Modifications

A component may be instantiated multiple times in different locations of the model. Enclosed components and primitive variables may thus be instantiated multiple times in different environments. The flattening process need therefore compute the set of primitive variables in all instantiated components. A simple way to implement flattening is to recursively flatten composite components and to parametrize the flattening process by a name prefix representing the environment in which the component is instantiated. Consider the code below. Each `ClassDecl` iterates over its components and equations, and flattens them with the current environment as a parameter. Another parameter holds a container for the resulting flattened code. Each `ComponentDecl` updates the environment by extending the prefix with its name. This particular strategy generates name prefixes that are not only unique but also provide traceability back to the original model. Each `ComponentDecl` of primitive kind generates a new variable in the resulting container, and non primitive components are flattened recursively with the updated environment.

```
public void ClassDecl.flatten(Collection res, String env) {
  for(Iterator iter = components().iterator(); iter.hasNext(); ) {
    ComponentDecl d = (ComponentDecl)iter.next();
    d.flatten(res, env);
  }
  // similar for equations
}
public void ComponentDecl.flatten(Collection res, String env) {
  String newEnv = env + getName();
  if(myClassDecl().isPrimitive())
    res.add(new Variable(newEnv, ... ));
  else
    myClassDecl().flatten(res, newEnv);
}
```

4.3.2 Flattening with modifications

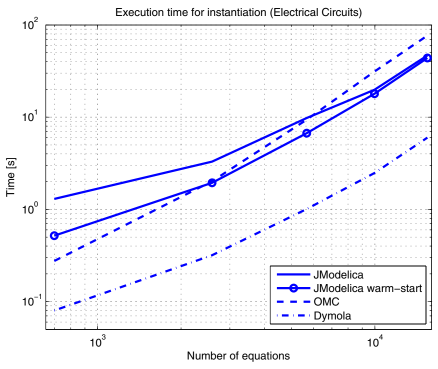
In the previous section we modelled the environment in which each class is instantiated with a string. However, the environment needed in Modelica is more complex due to the use of *modifications*. A modification changes properties of an enclosed component, and the effect of these modifications need therefore be taken into account when flattening a component. These effects can be quite complex, requiring highly context-sensitive computations, when using a special kind of modification called *redeclarations*. Such modifications may redeclare classes and components of an existing class. This allows for a kind of generic classes for which, for instance, the subtype relation described in Section 4.2.2 must not be violated. Another challenge is that Modelica permits several modifications on the the same element, as long as they are specified at different levels in the hierarchical model. In order to determine which modification to apply, the merge operation should implement the rule stating that outer modifications override inner modifications.

The environment in the PM compiler is modelled as a list of references to enclosing components and an ordered list of references to modifications. By inserting modifications at the end of the list, outer modifications will shadow inner modifications applicable to the same element. References allow us to query enclosing components and modifications to acquire more information about their context. Each component computes a new environment but in a more complex way than in Section 4.3.1. Modifications in the enclosing environment and locally declared modifications are merged into a new environment, which is used for the recursive call to flatten. The use of references in the environment allow us to query attributes in the referred modifications, e.g., use name binding to determine whether a particular modification is applicable to a component.

The flattening framework consists of 460 lines of, mainly imperative, code.

5 Test Results

The correctness and performance of the instantiation procedure of the JModelica compiler have been evaluated and compared to two other Modelica tools: the OpenModelica compiler and Dymola. Two model structures have been constructed to be used for benchmarking. The first model consists of a network of electrical components, and the second model consists of hierarchically nested classes. Models with increasing complexity were automatically generated in order to compare the execution time for small as well as large models. Correctness was verified by automatic comparison of the resulting flat descriptions from the three tools. The flat descriptions produced by the tools were identical in all cases. The execution times for the instantiation procedure of the three tools are compared in Figure 5. As can be seen, Dymola outperforms JModelica and OpenModelica in all cases. This should be no surprise, since Dymola is a highly optimized commercial product. For models of small and moderate size, the OpenModelica compiler is faster than JModelica. However, as model complexity increases, the JModelica compiler is faster than the OpenModelica compiler. Since it is well known that Java programs often execute slowly initially, execution times for the JModelica compiler was recorded also when the program was warm-started. That is, two models were parsed and instantiated in sequence as a single Java-program. By measuring the execution time of the second instantiation, the time associated with the start-up of the Java virtual machine and the initial dynamic compilation is not included in the result. As can be seen in Figure 5, the execution time of the JModelica compiler is significantly decreased for the small models, and is on par with those of the OpenModelica compiler. All benchmarks were performed on an Intel Core 2 Duo E6400 system equipped with 4Gb of memory. The JModelica compiler was run on the Java HotSpot Client VM version 1.5.0 with a 512 Mb sized heap.



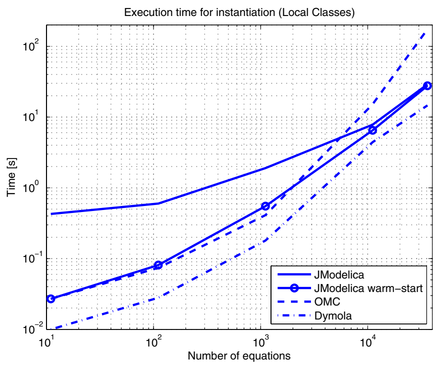


Figure 5. Benchmark results.

6 Conclusions and future work

In this paper, we have reported design concepts and experiences from a project where a Modelica compiler, JModelica, is being developed using the compiler tool JastAdd. It has been shown how complex semantic rules in the Modelica language can be implemented in a compact, modular and easy to understand manner. In particular, name analysis, type analysis and the flattening procedure have been treated.

The solutions presented in this paper all rely heavily on reference attributes and object-orientation. References allow us to remotely access any desired context while object-orientation provides abstraction for those contexts by not exposing too much details. Complex context-sensitive computations are easily modularized using these techniques and we were able to reuse designs from previous compilers, in particular, *delegating lookup attributes* for name analysis and *double dispatching attributes* for subtype comparisons. While these techniques were originally developed for building a Java compiler, they proved sufficiently versatile for adapting to the specific compilation issues in Modelica, like the modification construct, and the structural subtyping.

The use of a declarative approach has also proved to be beneficial for extensibility purposes, which is an important goal for our future work. For example, the basic name analysis for OO was possible to extend in a modular fashion to handle also the modification construct. Also, the implementation of flattening was possible to write in a simple and modular way, using the reference attributes defined in the name and type analysis as an API.

Benchmark results where the current implementation of the JModelica compiler is compared to two other Modelica tools are reported. In terms of execution times, the JModelica compiler performs well, especially for larger models.

A striking observation can be made regarding the development time of the compiler. As of November 2006, approximately 3 man-months, have been spent on implementation. Since the current version of JModelica implements several important features of Modelica, it is our experience that JastAdd is very well suited for rapid development of compilers.

Currently, the JModelica compiler is capable of parsing full Modelica 2.2, but has flattening support for only a subset of the language. This subset is growing, and currently includes generation of connection equations, merging of modifiers, lookup using import-clauses and treatment of short class definitions (alias classes). In addition, the compiler includes a module for error checking.

The JModelica compiler will serve as a key building block in the extensible Modelica environment which is under development. In particular, future research will focus on how complementing high level descriptions can be formulated in order to offer a natural interface to existing Modelica models.

References

[1] K.J. Åström, H. Elmqvist, and S.E. Mattsson. Evolution of continuous-time modeling and simulation. In *Proceedings of the 12th European Simulation Multiconference, ESM'98*, pages 9–18, Manchester, UK, June 1998. Society for Computer Simulation International.

[2] M. Andersson, S.E. Mattsson, D.M. Brück, and T Schönthal. OmSim—An integrated environment for object-oriented modelling and simulation. In *Proceedings of the IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, CACSD'94*, pages 285–290, Tucson, Arizona, March 1994.

[3] D. Broman, P. Fritzson, and S. Furic. Types in the modelica language. In *Proceedings of the 5th International Modelica Conference*, September 2006.

[4] Dynasim AB, 2006. http://www.dynasim.se.

[5] T. Ekman and G. Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer-Verlag, 2004.

[6] T. Ekman and G. Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005*, volume 4143 of *LNCS*. Springer-Verlag, 2006.

[7] T. Ekman. *Extensible Compiler Construction.* PhD thesis, Lund University, Sweden, June 2006.

[8] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN symposium on Compiler contruction*, pages 85–98. ACM Press, 1986.

[9] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* John Wiley & Sons, 2004.

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

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

[12] JastAdd, 2006. http://jastadd.cs.lth.se/web/.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355. Springer-Verlag, 2001.

[14] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (March 1971).

[15] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[16] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Electr. Notes Theor. Comput. Sci.*, 82(3), 2003.

[17] The modelica association, 2006. http://www.modelica.org.

[18] The OpenModelica Project, 2006. http://www.ida.liu.se/~pelab/modelica/OpenModelica.html.

[19] Picomodelica, 2006. http://www.control.lth.se/user/jakesson/picomodelica/.

[20] The Modelica Association. Modelica – a unified object-oriented language for physical systems modeling, language specification, version 2.2. Technical report, Modelica Association, 2005.

[21] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of PLDI 1989*, pages 131–145. ACM Press, 1989.