Science of Computer Programming 75 (2010) 21–38

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Implementation of a Modelica compiler using JastAdd attribute grammars

Johan Åkesson [a,*], Torbjörn Ekman [b], Görel Hedin [c]

[a] Department of Automatic Control, Lund University, Lund, Sweden
[b] Computing Laboratory, University of Oxford, Oxford, United Kingdom
[c] Department of Computer Science, Lund University, Lund, Sweden

ARTICLE INFO

Article history:
Received 2 May 2008
Received in revised form 14 February 2009
Accepted 9 July 2009
Available online 30 July 2009

Keywords:
Compiler construction
JastAdd
Modelica
Reference attributed grammars

ABSTRACT

We have implemented a compiler for key parts of Modelica, an object-oriented language supporting equation-based modeling and simulation of complex physical systems. The compiler is extensible, to support experiments with emerging tools for physical models. To achieve extensibility, the implementation is done declaratively in JastAdd, a metacompilation system supporting modern attribute grammar mechanisms such as reference attributes and nonterminal attributes.

This paper reports on experiences from this implementation. For name and type analyses, we illustrate how declarative design strategies, originally developed for a Java compiler, could be reused to support Modelica's advanced features of multiple inheritance and structural subtyping. Furthermore, we present new general design strategies for declarative generation of target ASTs from source ASTs. We illustrate how these strategies are used to resolve a generics-like feature of Modelica called *modifications*, and to support *flattening*, a fundamental part of Modelica compilation. To validate that the approach is practical, we have compared the execution speed of our compiler to two existing Modelica compilers.

1. Introduction

The Modelica language is widely used in industry for modeling and simulation of physical systems [1]. Application areas include industrial robotics, automotive systems 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, 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

* Corresponding address: Department of Automatic Control, Lund University, Box 118, SE-221 00 Lund, Sweden. Tel.: +46 46 222 87 80; fax: +46 46 138118.
E-mail addresses: jakesson@control.lth.se (J. Åkesson), torbjorn@comlab.ox.ac.uk (T. Ekman), gorel@cs.lth.se (G. Hedin).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.07.003

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 to 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. Modularity in this context is important in order to separate the core physical model from different model usages.

As a first 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 metacompilation tool JastAdd [2,3]. This tool is based on attribute grammars and supports several modern features such as reference attributes and nonterminal attributes, in order to support building extensible compilers. The choice of JastAdd is natural, since one of the primary targets of the JModelica compiler is to create an extensible Modelica environment. Extensibility is supported by JastAdd, both at the language level and at the implementation level, which makes JastAdd particularly well suited as compiler construction platform in this project.

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 the *flattening* of Modelica models. Flattening is the process of eliminating hierarchical constructs, like subclassing and object composition, and producing a flat set of variables and equations that can be interfaced with numerical algorithms.

In our implementation of the JModelica compiler, we have reused several JastAdd design strategies that were originally developed for Java name and type analysis [4,3], thereby illustrating the generality of these strategies. To solve the Modelica-specific problem of flattening, we have developed two new design strategies, *reference coupled ASTs* and *declarative AST generation*. These design strategies are general and can be applied to the general problem of how to declaratively create a generated target AST that has links back to the original source AST.

Modelica is a large and complex language. In order to demonstrate the main design principles of the JModelica compiler, a small subset of Modelica, entitled PicoModelica has been defined. Strategies and methods will be demonstrated in this paper using PicoModelica, for reasons of clarity and compactness. Details on PicoModelica can be found in [5].

The paper is organized as follows. Section 2 gives a background on the Modelica language. Section 3 contains an overview of the compiler construction tool JastAdd. Section 4 discusses how existing JastAdd strategies for name and type analysis have been reused and extended for Modelica. Section 5 describes the new technique of declaratively generating target ASTs based on a source AST. Sections 6 and 7 give a detailed treatment of how our new design strategies are applied to flattening. In Section 8 the performance of the JModelica compiler is compared to two other Modelica tools. The paper ends with a Section 9 with conclusions and future directions.

2. 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 modeling and simulation, see [6]. 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 the simulation of complex heterogeneous systems difficult. Based on experiences from designing other modeling languages, notably Omola, [7], the fundamental concepts of *object-orientation* and *declarative programming* were adopted. The latest version of the Modelica specification, 3.1, (see [8]) was released in May 2009.

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 differential and algebraic equations that typically capture laws of nature. Classes and multiple inheritance are used for abstracting and specializing the components and their behavior.

2.1. Bouncing ball example

Fig. 1 shows a simple Modelica example. A class BouncingBall is defined to model the behavior of bouncing balls, using Newton's second law. The example illustrates several key features of Modelica:

Classes and components The description of the bouncing ball is encapsulated in a class, using the keyword `model`. It 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` has a set of attributes, such as start value and unit. When declaring components, their subcomponents or attributes can be modified. For instance, in the declaration of `pos`, the start value is set to 1. This mechanism is called *value modification*. In more elaborate cases, modifications can be used to express generics. This is done by applying *structural modifications* that specialize the type of a subcomponent,

```
model 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;
```

Fig. 1. A class modeling bouncing balls.

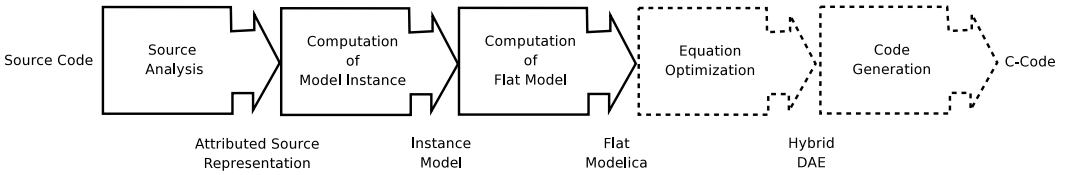


Fig. 2. The Modelica translation process.

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].

2.2. Compiling Modelica models

While object-oriented equation-based modeling in Modelica is beneficial for the modeler, the source code is not immediately suited for use with numerical algorithms. Typically, algorithms for simulation or optimization require a model to be represented in a form which is closely related to its mathematical definition, which is in the Modelica context usually referred to as the underlying hybrid differential algebraic equation (DAE). This model representation is flat, in the sense that it contains variables and equations, but no structural entities such as classes or components. The process of transforming a Modelica model into a flat representation is called *flattening*.

A Modelica program differs from many traditional programming languages in that it is not intended to be executed under the assumption of a program counter, which defines the point of execution. Neither is there a stack or heap. Rather, a Modelica model is defined by a static composition of components. The entire concept of nested components can therefore be eliminated at compile-time. All hierarchical constructs are then removed and the result is a flat description consisting of a set of uniquely named variables of primitive type, possibly arrays, and a set of equations and algorithms that operate on primitive values and variables only.

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 Fig. 2. In the first step, the source code is analyzed by means of conventional compiler mechanisms, such as name and type analysis. Typically, identifiers are bound to declarations and type compatibility is checked. The result of this step is an *attributed source representation*. The second step in a typical Modelica compilation procedure, is the computation of an *model instance* of a given model class, i.e., its nested component structure. In the third step, the Modelica code is *flattened*, eliminating all component, class, and inheritance structures. 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 qualified names are used for variables, indicating the path of the corresponding variable. Consider the following model which contains two components of the `BouncingBall` class in Fig. 1:

```
model BBex
  BouncingBall eBall;  // Ball on earth
  BouncingBall mBall(g=1.62); // Ball on the moon
end BBex;
```

```
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;
```

Fig. 3. A flat Modelica description.

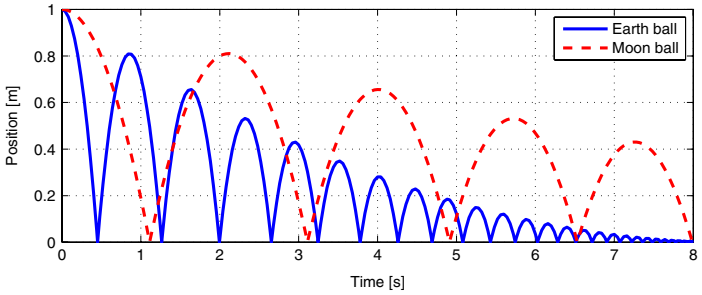


Fig. 4. Position as a function of time for the `BouncingBall` example.

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, in this case `g=9.81`, as defined in the BouncingBall class. The result of the flattening operation is shown in Fig. 3. The keyword `fclass` is used here to denote a flat Modelica model. In addition, Fig. 4 shows the positions of the two balls as a function of time.

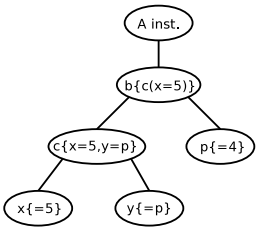
In the fourth step, the equations are sorted, analyzed and optimized so that they can be used with numerical software. The output of this step is referred to as a hybrid DAE, which is a mathematical object. The hybrid DAE also represents a generic mathematical description of the original Modelica model, 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 Fig. 4 are examples of a simulation result. The main focus of this paper is the first three steps in the compilation process, namely source analysis, model instance computation, and flattening of Modelica models. While the source analysis for Modelica has many features in common with that of conventional programming languages, model instance computation and flattening are Modelica-specific and challenging problems. Accordingly, the latter two steps will be given the most attention in this paper.

There are several compilers available for Modelica: one commercial product is Dymola from Dynasim, [10] and an open-source alternative is OpenModelica, [11]. Our own prototype compiler, JModelica, currently performs flattening for a subset of Modelica. In order to validate the correctness of the results, the flat descriptions produced by JModelica and Dymola were compared and verified. The compilation speed of JModelica is compared to both Dymola and OpenModelica in Section 8.

Source code

```
model A
  model B
    model C
      Real x=1;
      Real y=2;
    end C;
    parameter Real p=4;
    C c(x=3, y=p);
  end B;
  B b(c(x=5));
end A;
```

Instance AST



Flat model

```
fclass A
  parameter Real b.p=4;
  Real b.c.x=5;
  Real b.c.y=b.p;
end A
```

Fig. 5. The source code (left) of a Modelica model illustrating merging of modifications, the corresponding instance AST (middle) and the resulting flat Modelica model (right).

2.3. Flattening and modifications

Flattening of Modelica models is challenging due to *modifications*, which may change the value of a primitive variable or specialize the type of a subcomponent. In the former case the term *value modification* will be used and the latter case will be referred to as a *structural modification*. When flattening a given instance, its complete *modification environment*, consisting of a set of modifications, must be considered. How to represent and compute modification environments is a key issue when designing a flattening algorithm. In this paper we consider only value modifications. The extension to handle structural modifications is straightforward, and is discussed in [5].

Different instances of the same class may have different modification environments, depending on their context of enclosing instances. The modification environment is built by *merging* applicable modifications along the instance hierarchy. The Modelica specification states that *outer* modifications override *inner* modifications. That is, modifications applicable at an enclosing instance will override modifications defined for the instance itself.

To illustrate the concepts of environments and merging, consider the example in Fig. 5. The left column shows the source code, the middle column shows the instance hierarchy with modification environments, and the right column the flattened code for class `A`. The `A` instance has no modification environment since it is the root. The `b` instance has the modification environment `{c(x=5)}`, due to the declaration of `b`. Inside `b` there is a `c` instance where several modifications are applicable: `{x=3, y=p}` due to its declaration, and `{x=5}` due to its enclosing instance. Merging these modifications, where outer modifications take precedence, results in the environment `{x=5, y=p}`. For `x`, the modification `{=1}` from its declaration is merged with `{=5}` from its enclosing instance, resulting in the environment `{=5}`. Similarly for `y`, the resulting environment is `{=p}`.

Given the instance hierarchy and the modification environments, it is straightforward to produce the flattened code, shown to the right in the figure. To be able to compute the qualified names of expressions, like `p`, the modification environment also contains information about what instances the different modifications emanate from. This will be discussed in more detail in Section 6.

In order to illustrate our compiler design, we have defined a smaller language, PicoModelica, which is much simpler than Modelica, yet captures its characteristic advanced features, such as classes, components, inheritance, equations, and modifications. The source code for the PicoModelica compiler is available at, [12]. The detailed examples in this paper are taken from PicoModelica, but essentially the same design is used in the JModelica compiler.

3. JastAdd attribute grammars

The JastAdd way of building a compiler is to represent the program as an attributed abstract syntax tree (AST), where the attributes represent properties of individual syntax tree nodes. The (unattributed) AST is built by a parser, and attributes are defined using attribute grammars. The AST is represented as a tree of Java objects, and the attributes as methods on those objects. Attribute evaluation is implicit: when accessing an attribute (by calling its method), the attribute is automatically evaluated. Many attributes are cached so that subsequent accesses to the same attribute can return the value directly rather than to reevaluate it. The compiler writer can thus think of the AST as being fully attributed as soon as the parser has constructed the tree, whereas in reality the attributes are evaluated on demand.

The JastAdd language builds on Java, and hence inherits Java's object oriented features for modularization and reuse. The 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 [13], *synthesized* and *inherited* attributes are used for propagating data upwards and downwards in the AST. In contrast to Knuth-style attribute grammars, 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 [14]. Typically, reference attributes are used for representing name bindings, e.g., from variable use to declaration, from class to superclass,

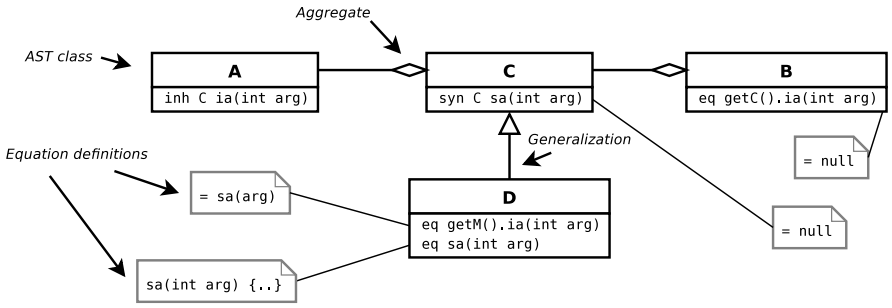


Fig. 6. AST classes, attributes and equations in extended UML notation.

etc. The attribute declarations and equations are specified using the keywords `syn`, `inh`, and `eq`. They are declared in aspect modules using *inter-type declarations* like in AspectJ [15], i.e., lexically separate from the classes they belong to. For instance, `syn A.x;` declares the synthesized attribute `x` of class `A`.

Another central feature of JastAdd is *parametrized attributes* that can take other attributes or expressions as parameters. The combination of reference attributes and parameterized attributes allow computations to be expressed at a high level, delegating subproblems to suitable nodes in the AST. The design strategies used for both name analysis and type analysis rely on this combination. For name analysis, the lookup of identifiers is delegated from one block to another, according to the scope rules. For example, from inner blocks to outer blocks, and from subclasses to superclasses. For type analysis, comparison of the types of two expressions is delegated to their corresponding type declarations. This is done through the double dispatch pattern [16]: the type comparison is first delegated to one of the type declarations, which in turn delegates it to the other type declaration. The application of these strategies to Modelica is discussed in Section 4.

Fig. 6 shows examples of some attributes and equations using an extension of UML [17] as a graphical notation for JastAdd code [3]. Attribute declarations are displayed as class methods, carrying one of the prefixes `inh` or `syn`. Equations are shown using a similar syntax, with the prefix `eq`, but for brevity without the return type. The right-hand side of equation declarations are shown in note boxes attached to the appropriate class. For example, the class `A` has an inherited parameterized attribute `ia`, and the class `C` has a synthesized attribute `sa`, also parameterized. In addition, an equation is defined for `ia` in `B`, which is an ancestor node of `C`. In `D` which is a subclass to `C`, an equation for `sa` overrides the definition in `C`, similar to method overriding.

In addition to reference attributes and parameterized attributes, JastAdd supports several other attribution features. Of particular importance to this paper are *nonterminal attributes* and *intrinsic attributes*, discussed in more detail in Section 5.

4. Reusing design strategies for name and type analysis

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 strategy as in the JastAdd Java implementation [4], 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 [4] and in our JModelica compiler, the environment is instead defined as an inherited parameterized attribute with the typical signature `ComponentDecl lookupDecl(String)`. Equations implementing lookup delegate to other lookup attributes as needed, often via reference attributes.

Due to the use of lookup attributes, the definition of bindings becomes straightforward. Here, a simple equation for the reference attribute `myDecl` binds an access of a name to its appropriate declaration:

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

Because `lookupDecl` is an inherited attribute, each node that has an `Access` child must define its `lookupDecl` attribute. Consider a qualified access, modeled by a `Dot` node with the following abstract syntax definition:

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

`Dot` needs to define `lookupDecl` for both its children, which it can access using `getLeft()` and `getRight()`. To define `lookupDecl` for its `Right` child, the `Dot` node delegates to its `Left` child, as follows:

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

Here, the value of `getRight().lookupDecl` is found by accessing the reference attribute `myClass` of the `getLeft` child and delegating to `memberDecl`. 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.

4.2. 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 2.

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(String)`. This attribute, in turn, is defined as an inherited attribute of the AST class `Argument`, which is a superclass of `ComponentModification`. Since an `Argument` node may be the child of several different kinds of nodes, all these need to define the `lookupDeclInClass` 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 related Modelica 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.

4.3. Type analysis

Modelica has a structural type system in which relations such as type equivalence and subtype are determined by the structure of classes rather than through explicit declarations.

In this section, computation of the subtype relation for classes in PicoModelica is treated. The subtype relation is particularly important in a Modelica compiler, due to the complex type system and type rules. Subtype conditions are often checked when determining the validity of class or component redeclarations in parameterized classes. It is important to notice that the type system of Modelica differs from that of, for example, Java. In Java, a class uniquely defines a type, whereas in Modelica, the interface, i.e., the named public elements of a class, defines its type. The proposed implementation is based on the design strategy for type analysis presented in [18], but it has been adapted to the context of Modelica.

4.4. Computation of the subtype relation

The subtype relation is defined for classes, as well as for components in Modelica. In this section the implementation for subtype test of classes will be described, whereas subtype computation of components is covered in [5].

In PicoModelica, we represent types by subclasses to the abstract AST class `ClassDecl`. Model classes are represented by an AST class `Model`, and built-in reals by the AST class `RealClass`. The abstract grammar for this is sketched below:

```
abstract ClassDecl ::= ...;
Model : ClassDecl ::= ...;
RealClass : ClassDecl ::= ...;
```

`Models` are composites, i.e., they have component members, and their subtype relation is *structural*, i.e., depending on the component members rather than on the model class hierarchy: A model s is a subtype of another model 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.

The subtype test is defined in the compiler by the parameterized synthesized attribute `subType`. It is defined for the abstract AST class `ClassDecl` and evaluates to `true` if the argument is a supertype of the receiver:

```
syn ClassDecl.subtype(ClassDecl superType);
```

This general subtype test can be conveniently expressed using the double dispatch pattern [16], and is also used in [18]. Consider the equation for the attribute `subtype` defined for `Model`:

```
eq Model.subtype(ClassDecl superType) = superType.supertypeModel(this)
```

Using this strategy, the first dispatch is performed based on the type of the receiving object, in this case `Model`. The second dispatch is performed based on the type of the argument, i.e., the type of `superType`. The actual computation of the subtype relation is thus delegated to the attribute `supertypeModel`. Notice that the double dispatch pattern renders the actual computation to be performed for the supertype relation. The AST class`RealClass` also defines the attribute `subtype` which delegates computation to the attribute `supertypeReal` in the same manner.

In PicoModelica, a `Model` class cannot be a subtype of `RealClass`, or vice versa. It is therefore convenient to introduce default definitions of the attributes `supertypeModel` and `superTypeReal` for `ClassDecl`, which evaluate to false:

```
syn boolean ClassDecl.superTypeModel(Model subType) = false;
syn boolean ClassDecl.superTypeReal(RealClass subType) = false;
```

Two equations defining the attribute for `Model` and `RealClass` are then added, each performing the supertype test for two types of the same kind, i.e., for two models or for two reals:

```
eq Model.superTypeModel(Model superType) {
  // Check of subtype condition for members
}
eq RealClass.superTypeReal(RealClass superType) = true;
```

For a detailed treatment of the implementation of the type analysis framework, see [5].

5. A design strategy for generating target ASTs

The flattening problem in Modelica is very challenging to a compiler writer, due to the generics-like *modifications* construct. Rather than to try to generate the flat code directly from the source AST, we have found that the process can be substantially simplified by first generating an explicit instance AST, as indicated by our discussion around the example in Fig. 5. The source and instance ASTs are coupled: each instance node contains a reference back to the corresponding declaration in the source AST. This application illustrates a general problem: to generate a target AST from a source AST, keeping references from individual target AST nodes to corresponding source AST nodes. We refer to such coupled ASTs as *reference-coupled ASTs*, and to the references that go back from the target AST to the source AST as *inter-AST references*.

Reference-coupled ASTs eliminate the need to encode source AST information needed in later compilation steps in the target AST. Instead, the target AST can directly access the information in the source AST through the inter-AST references. For example, to compute the modification environment of an individual instance node, the modification constructs in the source AST can be accessed directly.

Actually, we have applied reference-coupled ASTs twice in our Modelica compiler: In addition to generating the instance AST from the source AST, we generate a flat AST from the instance AST. To generate the textual code from the flat AST is then a trivial matter. Fig. 7 illustrates the resulting structure.

5.1. Declarative generation of reference-coupled ASTs

In JastAdd, it is possible to define reference-coupled ASTs declaratively, using the features of *nonterminal attributes*, *fresh attributes*, and *intrinsic attributes*.

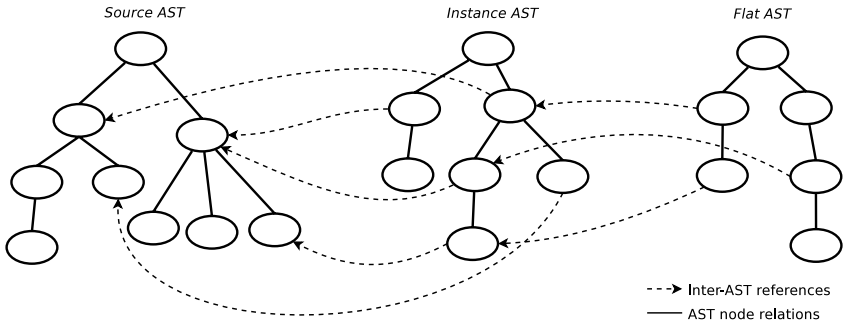


Fig. 7. Two applications of reference-coupled ASTs: The source AST is used to generate the instance AST, and the instance AST to generate the flat AST. The dashed arrows represent inter-AST references connecting target nodes to corresponding source nodes.

Nonterminal attributes (NTAs), also sometimes called higher-order attributes, are special reference attributes whose value is an AST [19]. Whereas ordinary reference attributes refer to existing nodes in the AST, the equation for an NTA *expands* the AST by computing a fresh AST subtree. This subtree can itself have attributes that are defined by other equations. NTAs thus give the possibility to define AST subtrees whose structure is context-dependent, and which could not therefore be constructed by the context-free parser. A typical use of NTAs is syntactic macro expansion.

Fresh attributes are used as help attributes for constructing NTAs. A fresh attribute returns a fresh AST subtree each time it is accessed. It is only allowed to be accessed by equations defining NTAs and other fresh attributes. It cannot be cached since it should always return a fresh subtree. By using fresh attributes, the equation for an NTA can delegate the construction of the new AST subtree to other AST nodes.

Intrinsic attributes are attributes that are defined at AST construction time rather than by an equation [20]. That is, their values are supplied by the parser or inside an NTA equation. A typical use of intrinsic attributes is for identifier strings and literal values, supplied by the parser. However, by combining reference attributes with intrinsic attributes, more interesting uses are possible. We can define an intrinsic attribute as a reference to an existing AST node, possibly in a different AST than the constructed AST. Intrinsic attributes can thus be used to represent *inter-AST references*.

To define a reference-coupled AST declaratively, an NTA is defined in a source AST node. The defining equation can use references in the source AST as intrinsic attributes of the new target AST, thereby establishing the inter-AST references. In the PicoModelica compiler, each class node has such an NTA, producing an instance AST for that class. A Modelica specification can contain hundreds of classes, but only one of them is selected by the user as the root class that should be flattened. Hence, demand evaluation is crucial for performance: only the instance AST of the selected class will actually be generated. Using this solution, the instance AST will technically be a part of the source AST. But conceptually, we may think of them as different ASTs.

5.2. Top-down AST generation

A generated target AST usually consists of many nodes, with many inter-AST references back to the source AST. Constructing it in one NTA equation would be very complex. However, if the construction of one target node depends only on its ancestor nodes (in addition to the source AST), the target AST can be represented as a tree of NTAs, and the generation can be defined by many small equations. Due to on-demand evaluation, the target AST will automatically grow top-down as it is traversed.

This ancestor dependency restriction holds for the Modelica instance ASTs: To construct an instance node, we need information about modification environments in its ancestor nodes, as well as information from the source AST. To define the instance AST, each instance node therefore defines all its children as a list of NTAs, each defined by an equation. Fig. 8 illustrates such top-down AST generation.

6. Generating the instance AST

The goal of generating the instance AST is to resolve Modelica modifications, to prepare for flattening. We will now describe how the instance AST is constructed, using the technique for top-down declarative generation of reference-coupled ASTs, as introduced in the previous section.

The instance AST is essentially a tree of instances of model classes. The root depends on which model class the user chooses to flatten. The structure of the instance tree is given by unfolding the class structure, starting at the root and creating new subnodes for its subcomponents. But this process is complicated by the fact that structural modifications, where types are redeclared, can alter the structure of a declared component. Therefore, to decide the structure of a particular class instance, it is not sufficient to look locally at that class. We also need to know what modifications apply at that point

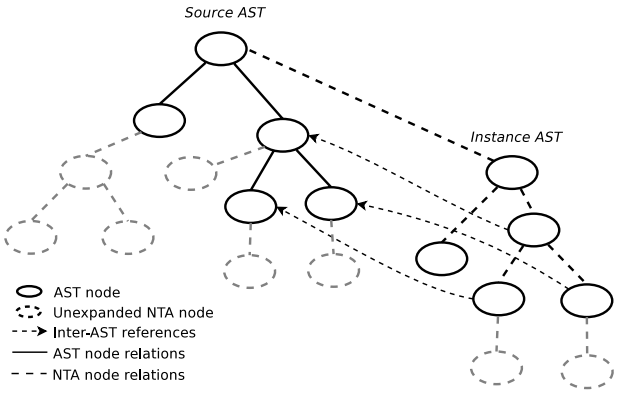


Fig. 8. Top-down generation of ASTs using NTAs.

in the instance AST. As a consequence, a particular component declaration may give rise to several different instance ASTs depending on the context in which it is flattened.

As was discussed in Section 2.3, the applicable modifications are available in ancestor nodes of an instance node. We can therefore construct the instance tree top-down, making use of existing instance nodes to decide the structure of a new node. This AST generation fits precisely with the declarative top-down generation using NTAs, as discussed in Section 5. For the purpose of brevity, we will limit our further discussion to value modifications, but extending the approach to cover also structural modifications is straightforward, given this top-down strategy of generation, and the details are available in [5].

6.1. The instance AST

A more precise characterization of the instance AST is that it contains one node for *each source node that may have a modification subtree*. For PicoModelica, this includes *component declarations*, (declarations of model instances and Real variables) and *extends clauses* (specifying superclasses of a given model).

We define the following abstract grammar for the PicoModelica instance AST:

```
abstract InstNode ::= /InstNode*/;
InstRoot : InstNode ::= <Model:Model>;
abstract InstComponent : InstNode ::= <ComponentDecl:ComponentDecl>;
InstModel : InstComponent;
InstReal : InstComponent;
InstExtends : InstNode ::= <ExtendsClause:ExtendsClause>;
```

The AST class names are prefixed with `Inst` to emphasize that they describe nodes in the instance AST, rather than in the source AST. The abstract class `InstNode` is a superclass of all nodes in the instance AST. It defines the structure of the AST using a list of children declared as an NTA, indicated by the enclosing slashes. Each node in the instance AST will thus have such an NTA list of children.

The instance AST is linked back to the source AST through inter-AST references: `Model`, `ComponentDecl`, and `ExtendsClause`. These are represented by intrinsic attributes, enclosed by angle brackets.[1]

The nodes in the instance AST are of different subclasses of `InstNode`. The root node is of the type `InstRoot`, and represents an instance of the model class to be flattened. Most other nodes represent different kinds of components: `InstModel`, representing a component of a `Model` type, and `InstReal`, representing primitive Real variables. Finally, `InstExtends` represents an extends clause of a model declaration. Fig. 9 shows the instance AST for the Modelica example in Fig. 5.

As mentioned earlier, all model declarations in the source AST can potentially be the root of an instance AST. It is the user that selects which model should be flattened. A root of an instance AST is therefore defined as an NTA of each model declaration in the source AST. This is done as follows:

```
Model ::= ... /InstRoot/;

syn Model.getInstRoot() = new InstRoot(this);
```

[1] The syntax <S:T> means that the name of the intrinsic attribute is S and that its type is T. Notice that the name and type strings may be equal, for example <T:T>. (Because intrinsic attributes are often strings, the shorthand <S> means an intrinsic string attribute named S.)

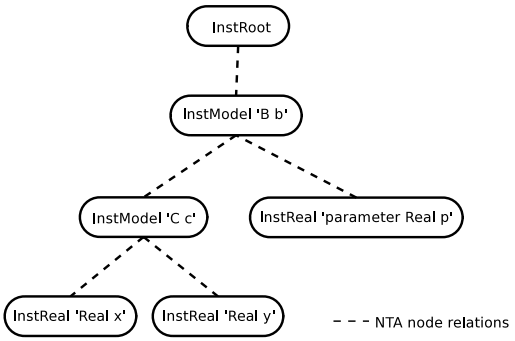


Fig. 9. Instance AST for the Modelica example in Fig. 5.

```
syn List InstNode.getInstNodeList() {
  List l = new List();

  // Add children corresponding to all components
  for (ComponentDecl cd : components()) {
    l.add(cd.newInstComponent(cd));
  }

  // Add children corresponding to all super classes
  for (ExtendsClause ec : supers()) {
    l.add(ec.newInstExtends());
  }
  return l;
}
```

Fig. 10. Equation defining the NTA `getInstNodeList`.

Here, the `Model` AST class declares `InstRoot`, the root of the instance AST, as an NTA. To define the NTA value, the accessor for the NTA, `getInstRoot()`, is declared as a synthesized attribute whose equation simply creates a fresh instance of `InstRoot`, supplying the `Model` itself (`this`) as the intrinsic attribute.

Starting in the source AST, if the `getInstRoot` NTA of a `Model` node is accessed, it will automatically be expanded, creating an `InstRoot` node. Similarly, the rest of the instance AST is created top-down by expanding the list NTA declared in each `InstNode`, as the tree is traversed. To create the appropriate instance AST nodes, relevant information in the source AST is accessed through the inter-AST references. Fig. 10 shows the equation defining the list NTA in `InstNode`.

Consider an `InstNode` object *n*. The equation in Fig. 10 defines `getInstNodeList` (its NTA attribute) as a list of freshly created `InstNode` objects, one for each component in *n*, and one for each extends clause.[2] The two attributes `components` and `supers` form a generic interface used in this computation. They are implemented by different equations in different `InstNode` subtypes, and use the inter-AST references to access relevant information in the source AST.

Fig. 11 shows the equations for the `components` attribute. Each equation delegates the computation to attributes in the source AST, also making use of attributes previously defined in the name and type analysis, such as `myClass`.

To actually create the `InstNode` objects, fresh attributes like `newInstComponent` are called from the NTA equation, as was shown in Fig. 10. Fig. 12 shows the definition of the fresh attributes. For components, an `InstModel` or an `InstReal` is created, depending on if the type of the component is a `Model` or a `RealClass`. This delegation of the AST subtree creation (from the NTA to the fresh attributes) is in essence an application of the factory pattern [21].

6.2. Modification environments

In order to be able to perform flattening, the instance trees need to include information about the *modification environment* for each node, as was exemplified in Fig. 5. In the full PicoModelica and JModelica implementations, the modification environments are also used by the factory methods to handle structural modifications, i.e., where the structure of a component instance depends on modifications.

The modifications are accessible from the instance tree through the inter-AST references, see Fig. 13. By searching in the correct order, the modification environment can be computed for each instance node. After several experiments, we have

[2] The equation uses method body syntax: instead of an ordinary expression, the right-hand side of the equation is formulated as a method body that returns a value. Imperative code is allowed inside such a method body as long as there are no externally visible side-effects, thus keeping with the declarativeness of attribute grammars.

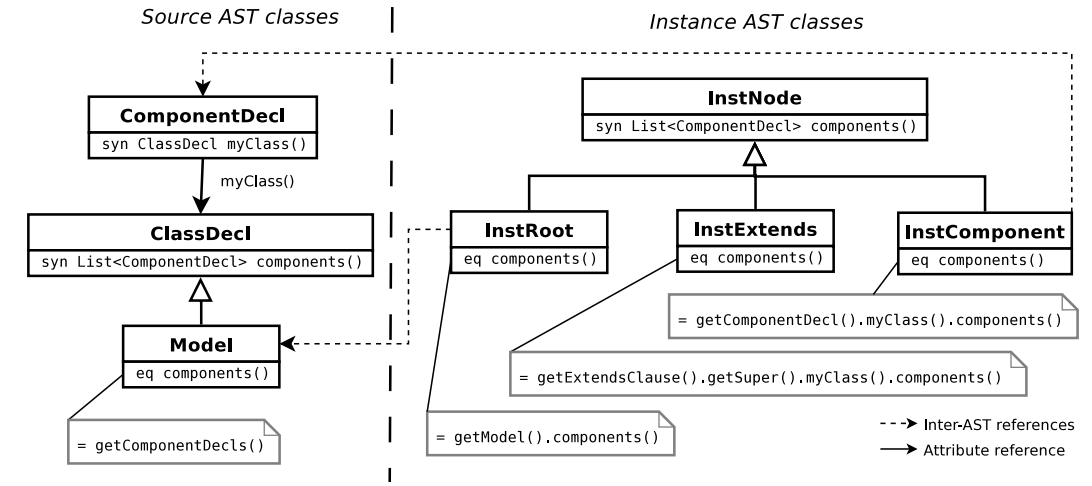


Fig. 11. The attribute `components` defined for `InstNode` gives access to all component declarations of the `InstNode`.

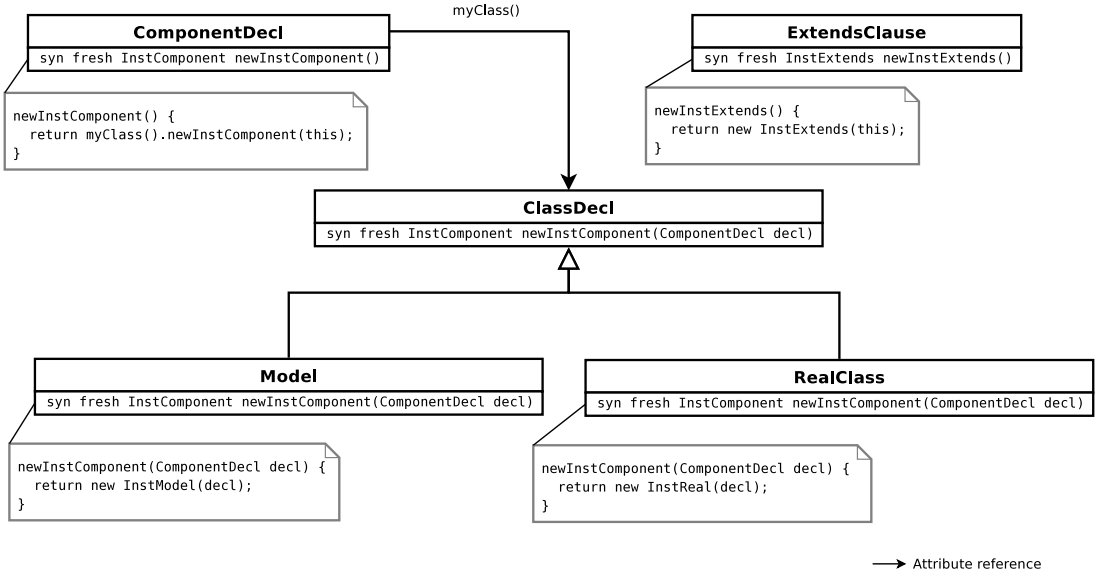


Fig. 12. Fresh attributes apply the factory pattern to create new `InstNodes` based on the corresponding classes in the source AST.

chosen to represent the modification environment for each instance node as an explicit data structure. We found that this has several advantages over searching: It simplifies the implementation, speeds up the compilation, and makes the compiler easier to extend with new language constructs.

Fig. 14 illustrates the chosen representation. Each instance node has a *modification environment* represented as a list of entries. Each entry has two references: one goes to the instance node from which the modification emanates, and the other goes to the appropriate modification subtree in the source AST. The list has similarities to a symbol table. It projects all applicable modifications for the instance node and upwards in the instance AST, onto a single data structure.

Fig. 15 shows the environments computed for the Modelica example in Fig. 5, replacing the references to modification subtrees by text, for better overview. As we can see from the example, an environment may contain several entries for the same name, in which case the earlier entry has precedence. To include the later entries may seem unnecessary when looking at a simple example like this. However, in more complex examples, all the entries may be needed. For example, if a node with the environment {`c(x=5), c(y=7)`} contains a `c` node, the `c` node will get an environment {`x=5, y=7`}.

During flattening, the appropriate value for a Real variable will be located in the first entry of the node's modification environment, as indicated by the arrows in the figure. For example, the appropriate modification of `y` is `=p`. The instance reference in the entry allows us to compute the qualified name for `p`. We see from this reference that the modification was introduced in the declaration of the `c` node. Those modifications have the same name context as the `c` declaration itself, i.e., `b`. So, the qualified name to be used in the flat code is `b.p`.

The modification environment of an instance node actually represents modifications that apply to its children. Consider an `InstComponent` *i* with the name *n* and parent *p*. The modification environment for *i* is constructed as follows:

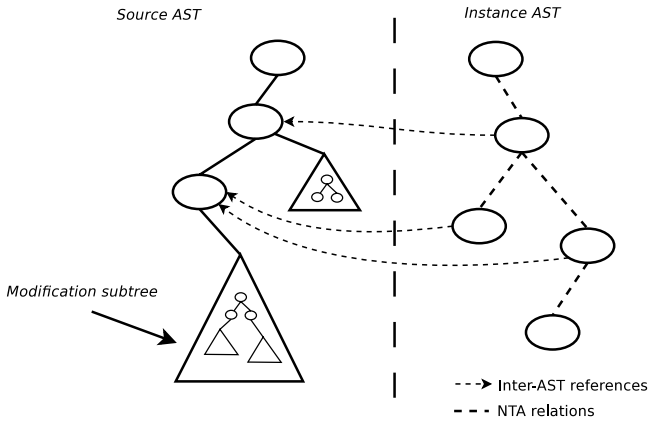


Fig. 13. Modification subtrees in the source AST can be accessed from the instance AST using inter-AST references.

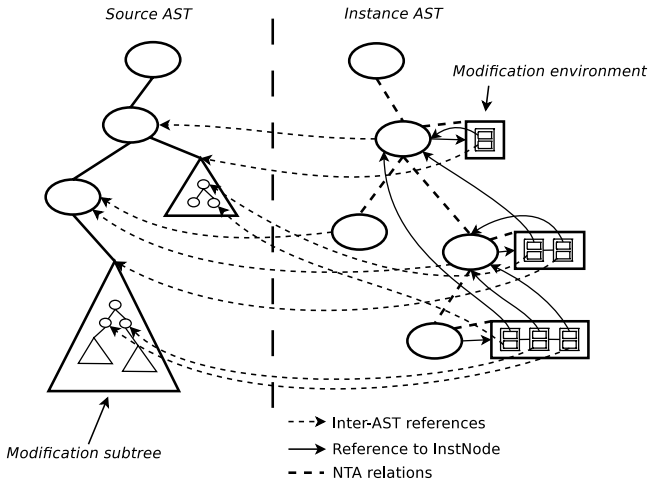


Fig. 14. Representation of modification environments in the instance AST and their relations to nodes in the source AST.

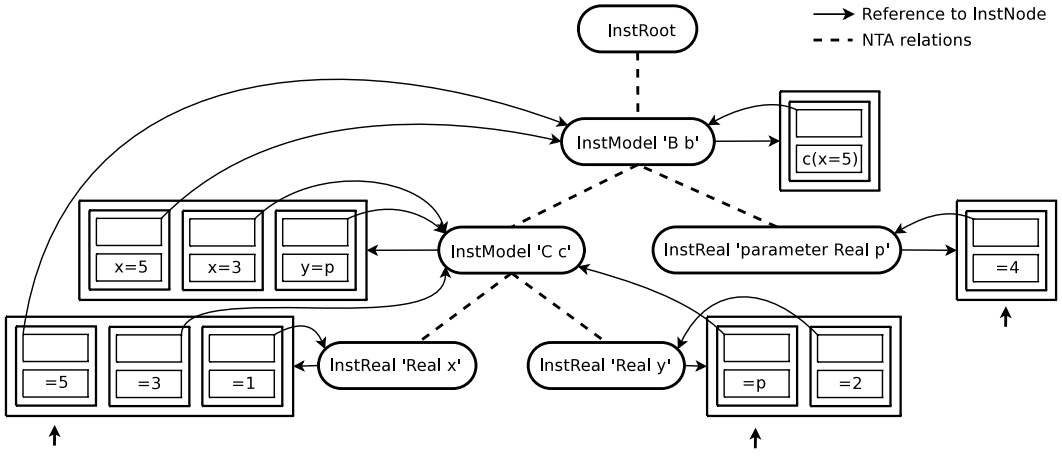


Fig. 15. Modification environments for the Modelica example in Fig. 5.

- Create a list l to hold the new modification environment for i.
- For each entry e in p's modification environment:
  - If e matches n, create new entries for each immediate subtree in e's modification tree. Set the instance reference in these entries to the same as in e. Add these entries to l.
- If i has a modification tree t in its corresponding source representation:
  - Create new entries for each top-level modification in t. Set the instance reference in these entries to i. Add these entries to l.

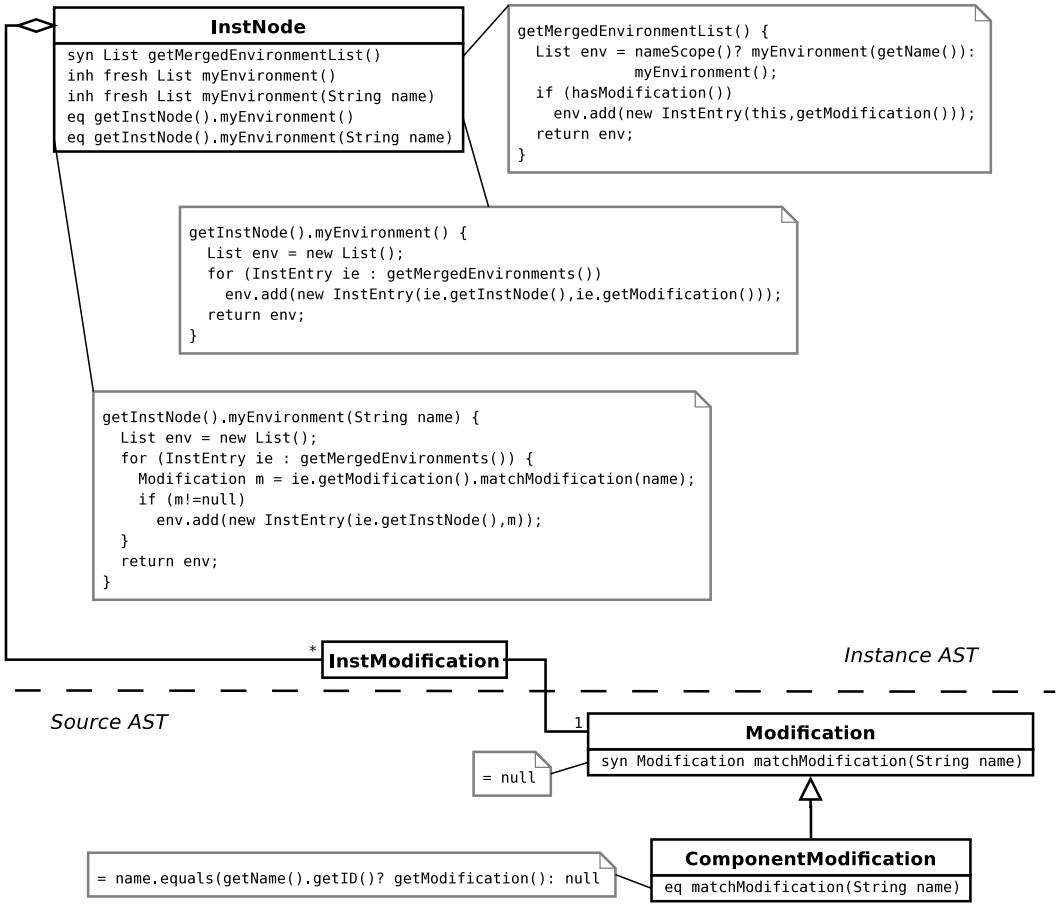


Fig. 16. The framework for computation of environments.

The procedure of constructing modification environments as described above is referred to as *merging* of modifications. As an example, consider computing the modification environment of the `x` node in Fig. 15. The modification environment of the parent consists of the three entries {`x=5, x=3, y=p`}. Two of these match `x`. Two new entries {`=5, =3`} are created where `x` is peeled off, and where the instance reference is set to the same object as in the original entries. The `x` declaration itself has a modification {`=1`} in the source program, and an entry for this is added last in the list, with the instance reference pointing to the `x` object itself.

6.3. Declarative construction of modification environments

From the algorithm above, we see that to compute the modification environment of an instance node, we only need information from the source AST and from ancestor instance nodes. This allows us to construct the modification environments using declarative top-down AST generation, representing them as NTAs.

We extend the abstract grammar for the instance AST as follows:

```
abstract InstNode ::= /InstNode*/ /MergedEnvironment:InstEntry*/;
InstEntry ::= <InstNode:InstNode> <Modification:Modification>;
```

Thus, for each `InstNode` there will be a new list NTA `MergedEnvironment` which contains entries (`InstEntry` objects). Each entry contains a reference to the appropriate `InstNode` and an inter-AST reference to the appropriate modification subtree in the source AST. Both these are represented as intrinsic attributes.

Fig. 16 shows the definition of the NTA. The top right box shows the main equation defining the corresponding synthesized attribute `getMergedEnvironmentList`. It first retrieves the inherited fresh attribute `myEnvironment`, i.e., the list of `InstEntry` AST nodes serving as the environment of the `InstNode` itself, and containing applicable outer modifications. Then it adds its own modification (if any), accessed through the `getModification` attribute which uses the inter-AST reference to locate the modification tree in the source AST (definition not shown).

The `myEnvironment` attribute comes in two variants: one with no arguments and one with a string argument representing a component name. In the first case, the resulting environment is a copy of the one defined by the

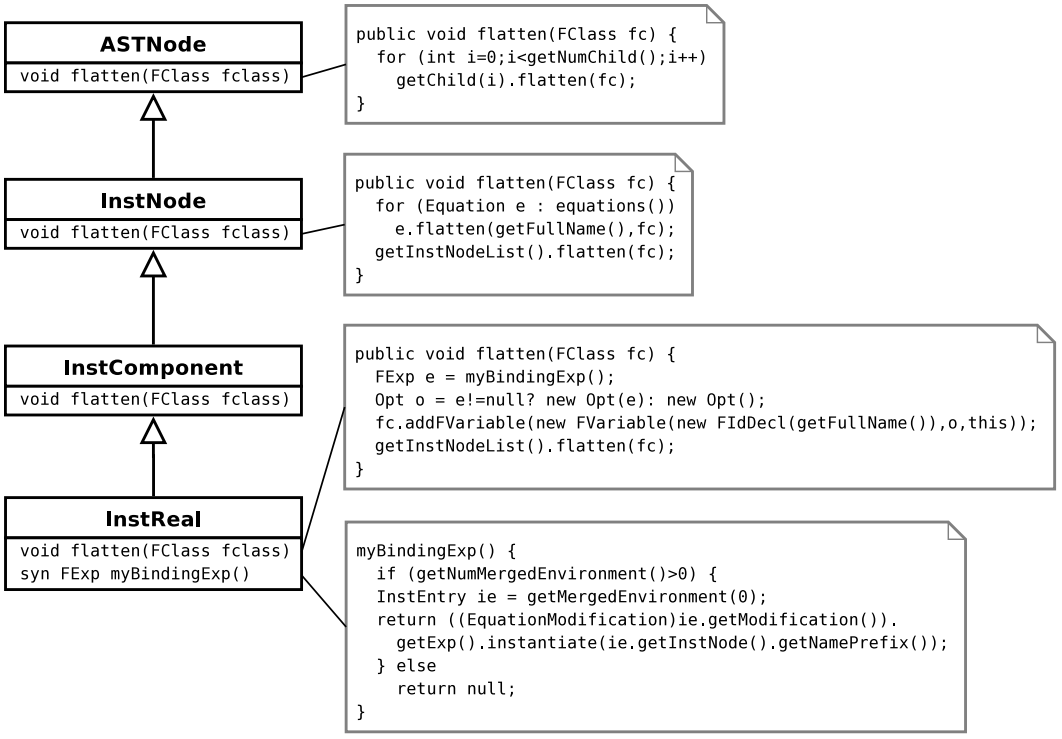


Fig. 17. The design of the flattening algorithm.

`getMergedEnvironmentList` attribute of the `InstNode`'s parent. In the second case, the resulting environment contains only modifications that match the supplied component name. The need for both versions stems from the fact that some instance nodes introduce a new name scope, for example `InstComponent`, whereas others, such as `InstExtends` do not.

The search for applicable modifications in the definition of the attribute `myEnvironment` relies on the synthesized attribute `matchModification`, which is defined for the source AST class `Modification`. If the string argument corresponding to a component name matches that of a `ComponentModification`, the modification subtree represented by the child modifications of this node is returned.

7. Generating the flat AST

Given that the instance AST has been constructed, flattening of a model can be performed by simply traversing the instance AST and collecting all equations and variables. The resulting model is represented by a new AST, referred to as the *flat* AST. In principle, it would be possible to use a subset of the source abstract grammar also to represent a flat model. However, since the flat representation constitutes an interface between the compiler front-end and different algorithms (symbolic and numerical), it is desirable to generate a new AST with a structure specifically designed for the purpose. Another reason for introducing a new abstract grammar for the flat model representation is that attributes defined for source AST classes may not be applicable in a flat context.

The flat abstract grammar consists in essence of a container class, `FClass`, which has one list of variables and one list of equations:

```
FClass ::= <Name:String> FVariable* FEquation*;
FVariable ::=  Name:FIdDecl [BindingExp:FExp] <InstReal:InstReal>;
FEquation ::= Left:FExp Right:FExp;
```

In addition, the flat abstract grammar contains classes which parallel the expression classes of the source abstract grammar. Notice the inter-AST reference `InstReal` defined for flat variables (`FVariable`). This gives access, for example, to the variable's modification environment in the instance AST, as well as its original declaration in the source AST.

The flat AST is created using normal methods that traverse the instance AST and fill in the appropriate variables and equations into an `FClass` object. The design of the algorithm is shown in Fig. 17. A generic traversal method `flatten(FClass fc)` is defined for the generic class `ASTNode`, which is the superclass of all AST classes. A specialization of the `flatten` method is defined for `InstNode`, in which all equations are retrieved from the source AST, flattened and added to the supplied `FClass`. A generic interface to the equations corresponding to an `InstNode` is provided by the attribute `equations`. This attribute is defined similarly to `components`, see Fig. 11.

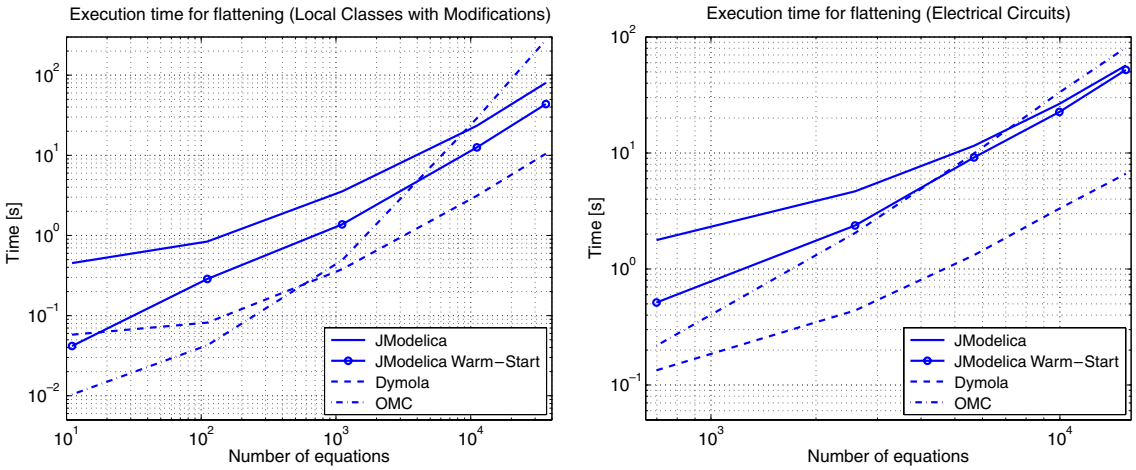


Fig. 18. Benchmark results.

Flattening of equations and expressions is straightforward–the flattened equation or expression is (almost) identical to the original one. The one complication is to compute the fully qualified names of variables. Given the nested component structure of the instance AST, qualified names of variables in a component are easily implemented by an inherited attribute that combines the ancestor names. When an expression is flattened, however, the correct environment must be used for variables inside, taking possible modifications into account. This is done by providing a *name prefix* to the flattening method for equations and expressions. If an expression contains an identifier, the supplied name prefix is simply concatenated with the name of the identifier.

A specialization of the flattening method is also provided for `InstReal`. In this case, a flat variable, represented by an `FVariable`, is added to the `FClass`. In addition, the environment of the primitive declaration must be checked for an applicable value modification, or *binding expression*. The synthesized attribute `myBindingExp()`, which is defined for `InstReal`, computes the correct expression. Notice that the binding expression is flattened using the name prefix of the `InstNode` which was originally associated with the modification subtree. Although subtle, this point is essential, in order for binding expressions given in modifications to have the correct prefix. In fact, the main reason for including a reference to the `InstNode` of a modification in the `InstEntry` class is to enable correct computation of the name prefix of expressions.

To summarize, the flattening algorithm is straightforward to implement, given that the instance AST, including modification environments, has been constructed. In essence, the flattening algorithm is implemented as a traversal of the instance AST, where all equations and primitive variables are collected.

8. Test results

The correctness and performance of the flattening algorithm of the JModelica compiler have been evaluated and compared to two other Modelica tools: the open-source compiler OpenModelica 1.4.3 (OMC) [11], and the commercial tool Dymola 6.0b [10]. Two model structures have been constructed to be used for benchmarking. The first model structure consists of a network of electrical components. The second model structure consists of hierarchically nested classes, where attributes of primitive variables are modified. Models with increasing complexity were automatically generated in order to compare the execution time for small as well as larger models. Correctness was verified by automatic comparison of the resulting flat descriptions from the three tools. The flat descriptions produced by the tools were equivalent in all cases. The execution times for the flattening procedure of the three tools are compared in Fig. 18. As can be seen, Dymola outperforms JModelica and OpenModelica, except for very small models. 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 were recorded also when the program was warm-started. That is, two models were parsed and flattened in sequence as a single Java-program. By measuring the execution time of the second flattening, the time associated with the start-up of the Java virtual machine and the initial dynamic Java compilation is not included in the result. As can be seen in Fig. 18, the execution time of the JModelica compiler is then significantly decreased for the small models.

It should be noticed that these benchmarks only test a small number of language constructs. Also, in the speed measurements Dymola also generates C code for simulation, whereas OMC and JModelica only performs flattening. Accordingly, the results cannot immediately be used to draw conclusions about the overall performance of the tested tools. However, the results indicate that the JModelica compiler executes at a reasonable speed, when compared to other Modelica tools.

All benchmarks were performed on an Intel Core 2 Duo E6420 system, equipped with 4Gb of memory, running Fedora Core 7. The JModelica compiler was run on the Java HotSpot Server VM version 1.5.0 with a 2.5 Gb sized heap. The need for this large heap stems from the fact that the compiled models are large—the largest model contains more than 30,000 variables and equations.

9. Conclusions and future work

In this paper, we have reported design strategies 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 have been able to reuse design strategies developed for a Java compiler [3]. The strategies for name analysis and type analysis have been reused and extended to cover semantic behavior in Modelica that has no equivalent in Java. In particular, the case of modifications has been considered in the name analysis framework. Also, it has been shown how computation of the subtype relation in Modelica can be implemented using the same strategy as in [3], even though the structural type system of Modelica differs significantly from the nominal type system of Java.

Two general strategies for compiler construction have been contributed in this paper. The strategy of *reference coupled ASTs* is useful when new ASTs are constructed based on a source AST. In the proposed strategy, inter-AST references from the target AST to the source AST enable convenient information access form the target AST. The strategy of *declarative AST generation* relies on nonterminal attributes, and enables compact and straightforward implementation. The AST generation is also performed on demand, so that only ASTs that are actually accessed are constructed.

The strategies of reference coupled ASTs and declarative AST generation have been applied in the implementation of the flattening algorithm in the compiler. The problem of flattening Modelica models is challenging, in particular due to modifications, and it has been shown how the proposed strategies enable compact, modular, and understandable compiler implementation. Based on the JModelica compiler, a modular extension, entitled Optimica, [5], dedicated to accommodate also dynamic optimization problems has been implemented. The extensibility capabilities of the compiler is analyzed and evaluated in [22] based on the experiences with implementing the Optimica extension.

Currently, the JModelica compiler is capable of parsing full Modelica 3.0, 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, treatment of short class definitions (alias classes) and some redeclare constructs. In addition, the compiler includes a module for error checking.

Benchmark results presented in the paper indicate that the proposed implementation strategies render a compiler with promising execution times. The compiler executes slower than the state of the art commercial software Dymola. However, it is on par with the open source software OpenModelica, in particular for larger models containing more than 10,000 variables and equations.

A striking observation can be made regarding the development time of the compiler. As of February 2009, approximately 9 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 the rapid development of compilers.

The applicability of the JModelica compiler has been evaluated in realistic applications, including optimal start-up of a plate reactor [23], race track optimization [24], and parameter optimization applied to vehicle models [25]. The JModelica compiler has also been used in teaching in a PhD level course on tools and languages for dynamic optimization.

Future research based on the JModelica platform include parallelization of optimization problems as well as deployment in industrial projects. The platform is available as open source software, see [12] for release information and [26] for an overview of the software platform.

References

[1] The Modelica Association, The Modelica Association Home Page, http://www.modelica.org. 2009.
[2] G. Hedin, E. Magnusson, JastAdd: An aspect-oriented compiler construction system, Science of Computer Programming 47 (1) (2003) 37–58.
[3] T. Ekman, G. Hedin, The JastAdd extensible Java compiler, in: Proceedings of OOPSLA 2007, 2007.
[4] T. Ekman, G. Hedin, Modular name analysis for Java using JastAdd, in: Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, in: LNCS, vol. 4143, Springer-Verlag, 2006.
[5] J. Åkesson, Tools and languages for optimization of large-scale systems, Ph.D. thesis, Department of Automatic Control, Lund University, Sweden, Nov. 2007.
[6] K.J. Åström, H. Elmqvist, S.E. Mattsson, Evolution of continuous-time modeling and simulation, in: Proceedings of the 12th European Simulation Multiconference, ESM'98, Society for Computer Simulation International, Manchester, UK, 1998, pp. 9–18.
[7] M. Andersson, S. Mattsson, D. Brück, 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, Tucson, Arizona, 1994, pp. 285–290.
[8] The Modelica Association, Modelica – a unified object-oriented language for physical systems modeling, language specification, version 3.1, Tech. rep., Modelica Association, 2009.
[9] P. Fritzson, Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, John Wiley & Sons, 2004.

[10] Dynasim AB, Dynasim AB Home Page, http://www.dynasim.se. 2009.
[11] PELAB, The OpenModelica Project, http://www.ida.liu.se/~pelab/modelica/OpenModelica.html. 2009.
[12] Modelon AB, JModelica Home Page, http://www.jmodelica.org. 2009.
[13] D.E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145. correction: Mathematical Systems Theory 5, 1, pp. 95-96 (March 1971.
[14] G. Hedin, Reference attributed grammars, Informatica (Slovenia) 24 (3) (2000) 301–317.
[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of Aspect J, in: LNCS, vol. 2072, 2001, pp. 327–355.
[16] D.H.H. Ingalls, A simple technique for handling multiple polymorphism, in: OOPLSA'86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, ACM Press, New York, NY, USA, 1986, pp. 347–349.
[17] Object Management Group, Unified Modeling Language, http://www.uml.org/. 2007.
[18] T. Ekman, Extensible Compiler Construction, Ph.D. thesis, Lund University, Sweden, Jun. 2006.
[19] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, Higher order attribute grammars, in: Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM Press, 1989, pp. 131–145.
[20] R. Farrow, Linguist-86: Yet another translator writing system based on attribute grammars, in: SIGPLAN '82: Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, ACM Press, New York, NY, USA, 1982, pp. 160–171.
[21] E.H. Gamma, R. Johnson, J.R. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[22] G. Hedin, J. Åkesson, T. Ekman, Extending languages by leveraging compilers - from Modelica to Optimica, IEEE Software, 2009 (submitted for publication).
[23] S. Haugwitz, J. Åkesson, P. Hagander, Dynamic start-up optimization of a plate reactor with uncertainties, Journal of Process Control (2008) doi:10.1016/j.jprocont.2008.07.005.
[24] H. Danielsson, Vehicle path optimisation, Master's Thesis ISRN LUTFD2/TFRT–5797–SE, Department of Automatic Control, Lund University, Sweden, Jun. 2007.
[25] H. Hultgren, H. Jonasson, Automatic calibration of vehicle models, Master's Thesis ISRN LUTFD2/TFRT–5794–SE, Department of Automatic Control, Lund University, Sweden, Jun. 2007.
[26] J. Åkesson, M. Gäfvert, H. Tummescheit, Jmodelica—an open source platform for optimization of modelica models, in: Proceedings of MATHMOD 2009 - 6th Vienna International Conference on Mathematical Modelling, TU Wien, Vienna, Austria, 2009.