

# Tutorial: Generating Language Tools with JastAdd

Görel Hedin

Lund University, Sweden  
gorel@cs.lth.se

Draft June 10, 2009. Preproceedings for GTTSE 2009.  
This version covers synthesized and inherited attributes, reference  
attributes, collection and circular attributes.

**Abstract.** JastAdd is an open-source system for generating compilers and other language-based tools. Its declarative specification language is based on attribute grammars and object-orientation. This allows tools to be implemented as composable extensible modules, as exemplified by JastAddJ, a complete extensible Java compiler. This tutorial gives an introduction to JastAdd and its underlying attribute grammar mechanisms, modularity mechanisms, and common design strategies. A simple state machine language is used as a running example.

**Key words:** attribute grammars, extensible language tools, reference attributes, object-oriented model

## 1 Introduction

JastAdd is a metacompilation system for generating language-based tools such as compilers, source code analyzers, and language-sensitive editing support. It is based on a combination of attribute grammars and object-orientation. The key feature of JastAdd is that it allows properties of abstract syntax tree nodes to be programmed declaratively. These properties, called *attributes*, can be simple values like integers, composite values like sets, and reference values which point to other nodes in the abstract syntax tree (AST). The reference values allows graph properties to be defined. For example, linking identifier uses to their declaration nodes, or representing call graphs and dataflow graphs. AST nodes are objects, and the resulting data structure, including attributes, becomes an object-oriented model, rather than only a simple syntax tree.

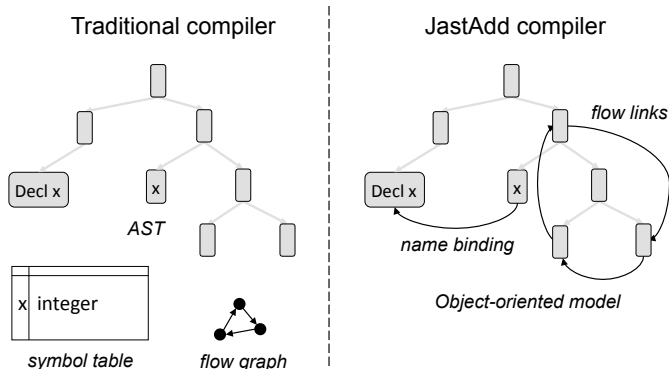
### 1.1 Object-oriented model

Figure 1 illustrates the difference from a traditional compiler where important data structures like symbol tables, flow graphs, etc., are typically separate from the AST. In JastAdd, these data structures are instead embedded in the AST, using attributes, resulting in an object-oriented model of the program. JastAdd is

integrated with Java, and the resulting model is implemented using Java classes, and the attributes form a method API to those classes.

Attributes are programmed declaratively, using *attribute grammars*: Their values are stated using *equations* that may access other attributes. Because of this declarative programming, the user does not have to worry about in what order to evaluate the attributes. The user simply builds an AST, typically using a parser, and all attributes will then automatically have the correct values according to their equations, and can be accessed using the method API. The actual evaluation of the attributes is carried out automatically and implicitly by the Jastadd system. The attribute grammars used in JastAdd go much beyond the classical attribute grammars defined by Knuth [Knu68], providing support for reference attributes [Hed00], parameterized attributes [Hed00, Ekm06], circular attributes [Far86, MH07], collection attributes [Boy96, MEH09], nonterminal attributes [VSK89], and rewrites [EH04].

An important consequence of the declarative programming is that the object-oriented model in Fig. 1 becomes extensible. The JastAdd user can simply add new attributes, equations, and syntax rules. This way, an existing structure can be extended, for example a name binding graph, or a completely new structure can be defined, for example a flow graph. This makes it easy to extend languages and to build new tools as extensions of existing ones.



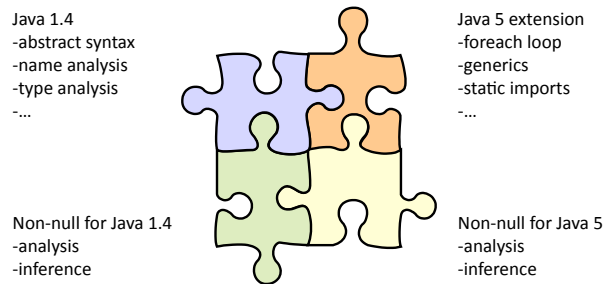
**Fig. 1.** In JastAdd, compilation data structures are embedded as attributes in the AST, resulting in an object-oriented model of the program.

## 1.2 Extensible languages and tools

The order of defining attributes and equations is irrelevant—their meaning is the same regardless of order. This allows the user to organize rules into modules arbitrarily, to form modules that are suitable for reuse and composition. Sometimes it is useful to organize modules based on compilation problems, like name

analysis, type analysis, dataflow analysis, etc. Other times it can be useful to organize modules based on language constructs, like support for new kinds of statements. As an example, in JastAddJ, an extensible Java compiler built using JastAdd [EH07a], both modularization principles are used, see Figure 2. Here, a basic compiler for Java 1.4 is modularized according to compiler phases: name analysis, type analysis, etc. In an extension to Java 5, modules for the new Java 5 language constructs are added, like the foreach loop, static imports, generics, etc. In yet further extensions, new computations are added, like non-null analysis [EH07b], separated into one module handling the Java 1.4 constructs, and another one handling the Java 5 constructs.

JastAdd has been used for implementing a variety of different languages, from small domain-specific languages like the state machine language that will be used in this tutorial, to full-blown general-purpose languages like Java. Because of the modularization support, it is particularly attractive to use JastAdd to build extensible languages and tools.



**Fig. 2.** Each component has modules containing abstract syntax rules, attributes, and equations. To construct a compiler supporting non-null analysis and inference for Java 5, all modules in the four components are used.

### 1.3 Tutorial outline

This tutorial gives an introduction to JastAdd and its underlying attribute grammar mechanisms, modularity mechanisms, and common design strategies. Section 2 presents a language for simple state machines that we will use as a running example. It is shown how to program towards the generated API for a language, constructing ASTs and using attributes. Basic attribution mechanisms are presented in Section 3, including synthesized and inherited attributes [Knu68], reference attributes [Hed00], and parameterized attributes [Hed00, Ekm06]. We show how name analysis can be implemented using these mechanisms. This section also briefly presents the underlying execution model.

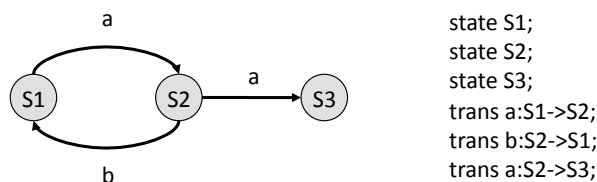
The two following sections present more advanced mechanisms. Section 4 discusses how to define composed properties like sets using *collection attributes*

[Boy96,MEH09]. These attributes are defined by the combination of contributions throughout an AST. We illustrate collection attributes by defining an explicit graph representation for the state machine, with explicit edges between state and transition objects. Section 5 discusses how recursive properties can be defined using *circular attributes* which are evaluated using fixed-point iteration [Far86,MH07]. This is illustrated by the computation of reachability sets for states.

Finally, Section 6 concludes the tutorial. Both the JastAdd metacompilation system and the JastAddJ extensible Java compiler are available as open-source tools at <http://jastadd.org>. The examples in this tutorial are tested on jastadd2.jar version R20080407.

## 2 Running example: A state machine language

As a running example, we will use a small state machine language. Figure 3 shows an example state machine depicted graphically, and a possible textual representation of the same machine, listing all its states and transitions.



**Fig. 3.** An example state machine and its textual representation.

### 2.1 Abstract grammar

In order to do various computations on the state machine, we would like to construct an object-oriented model of it that explicitly captures its graph properties. We can do this by first defining an abstract grammar that gives a tree representation corresponding to the textual representation, and then add reference attributes to represent the graph properties. Consider the abstract grammar in Fig. 4, written in JastAdd syntax.

This grammar models a state machine as a list of declarations which can be either states or transitions. From a grammar perspective, we can think of Declaration as a nonterminal and State and Transition as productions, and StateMachine serving the role as both nonterminal and production. In JastAdd, we view the grammar from an object-oriented perspective where all four are classes. Declaration is here an abstract class, and State and Transition are its subclasses. The entities Label, etc. represent tokens of type String, and can be thought of as fields of the corresponding classes.

```

StateMachine ::= Declaration*;
abstract Declaration;
State : Declaration ::= <Label:String>;
Transition : Declaration ::=
  <Label:String> <SourceLabel:String> <TargetLabel:String>;

```

Fig. 4. Abstract grammar for the state machine language

## 2.2 Attributing the AST

To obtain an explicit object-oriented model of the graph, we would like to link each state object to the transition objects that has that state object as its source, and to link each transition object to its target state object. This can be done using reference attributes. Figure 5 shows the resulting object-oriented model for the example machine in Figure 3. We see here how the edges between state and transition objects are embedded in the AST, using reference attributes. Given this object-oriented model, we might be interested in computing, for example, reachability. The set of reachable states could be represented as an attribute in each State object. In sections 3, 4, and 5 we will see how these attributes can be defined.

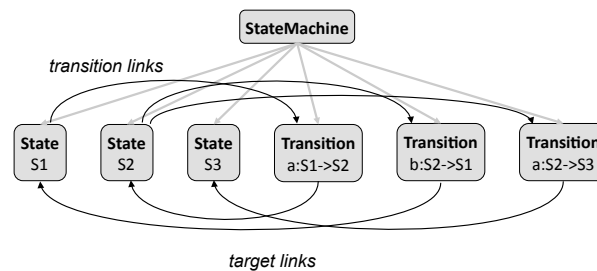


Fig. 5. The state machine graph is embedded in the object-oriented model.

*Exercise 1.* In Figure 5, the objects are laid out visually to emphasize the AST structure. Make a new drawing that instead emphasizes the state machine graph. Draw only the State and Transition objects and the links between them, mimicking the layout in Figure 3.

## 2.3 Building and using the AST

From the abstract grammar, JastAdd generates a Java API with constructors for building AST nodes and methods for traversing the AST. This API is furthermore augmented with methods for accessing the attributes. Figure 6 shows part

of the generated API for the state machine language, including the attributes `target`, `transitions`, and `reachable` that will be defined in the coming sections.

```

class StateMachine{
    StateMachine(); // AST construction
    void addDeclaration(Declaration node); // AST construction
    List<Declaration> getDeclarations(); // AST traversal
    Declaration getDeclaration(int i); // AST traversal
}

abstract class Declaration{
}

class State extends Declaration{
    State(String theLabel); // AST construction
    String getLabel(); // AST traversal
    Set<Transition> transitions(); // Attribute access
    Set<State> reachable(); // Attribute access
}

class Transition extends Declaration{
    Transition(String theLabel, theSourceLabel, theTargetLabel); // AST construction
    String getLabel(); // AST traversal
    String getSourceLabel(); // AST traversal
    String getTargetLabel(); // AST traversal
    State target(); // Attribute access
}

```

**Fig. 6.** API to the state machine model

Suppose we want to print out the reachable states for each state. For the small example in Figure 3, we would like to obtain the following output:

```

S1 can reach {S1, S2, S3}
S2 can reach {S1, S2, S3}
S3 can reach {}

```

meaning that all three states are reachable from S1 and S2, but no states are reachable from S3. To program this we would build an AST for the state machine, using the AST construction API, and then traverse the AST to print the information about reachable sets, using the traversal and attribute APIs. To program the traversal in a nice way we would like to add some ordinary Java methods to the AST classes. This can be done using a JastAdd *aspect* as shown in Fig. 7.

```

aspect PrintReachableInformationForStates {
    public void StateMachine.printReachable() {
        for (Declaration d : getDeclarations()) d.printReachable();
    }

    public void Declaration.printReachable() { }

    public void State.printReachable() {
        System.out.println(getLabel() + " can reach {" +
            listOfReachableStateLabels() + "}")
    }

    public String State.listOfReachableStateLabels() {
        boolean insideList = false;
        String result = "";
        for (State s : reachable()) {
            if (insideList) result.append(", ") else insideList = true;
            result.append(s.getLabel());
        }
        return result;
    }
}

```

**Fig. 7.** An aspect defining methods for printing the reachable information for each state.

The aspect uses *inter-type declarations* to add methods to existing classes. For example, the method `void StateMachine.printReachable() ...` means that the method `void printReachable() ...` is added to the class `StateMachine`.<sup>1</sup>

We can now write the main program that constructs the AST and prints the reachable information, as shown in Fig. 8. Here we have constructed the AST manually, using the construction API. An alternative would be to write a parser that uses the construction API in its semantic actions. Any Java-based parser generator could be used, provided it allows you to place arbitrary Java code in the semantic actions so the appropriate AST can be built. In earlier projects we have used, for example, the LR-based parser generators *CUP* and *beaver*, and the LL-based parser generator *JavaCC*. For parser generators that automatically provide their own AST representation, a possibility is to write a trivial visitor that traverses the parser-generator-specific AST and builds the corresponding JastAdd AST.

Notice that after building the AST, all the attributes are available automatically. For example, the `reachable` attribute used when executing the method

<sup>1</sup> This syntax for inter-type declarations is borrowed from AspectJ. Note, however, that JastAdd aspects support only static aspect-orientation in the form of these inter-type declarations. Dynamic aspect-orientation like pointcuts and advice are not supported.

```

class MainProgram {
    public void static main(String[] args) {
        // Construct the AST
        StateMachine m = new StateMachine();
        m.addDeclaration(new State("S1"));
        m.addDeclaration(new State("S2"));
        m.addDeclaration(new State("S3"));
        m.addDeclaration(new Transition("a", "S1", "S2"));
        m.addDeclaration(new Transition("b", "S2", "S1"));
        m.addDeclaration(new Transition("a", "S2", "S3"));

        // Print reachable information for all states
        m.printReachable();
    }
}

```

**Fig. 8.** A main program that builds an AST and then accesses attributes.

`printReachable` will have the correct value for each of the `State` objects—provided that we have defined it correctly using attributes and equations, as we will see in the following sections.

*Exercise 2.* Write an aspect that traverses a state machine and prints out information about each state, stating if it is on a cycle or not. Hint: You can use the call `s.contains(o)` to find out if the set `s` contains a reference to the object `o`. What is your output for the state machine in Fig. 3? What does your main program look like?

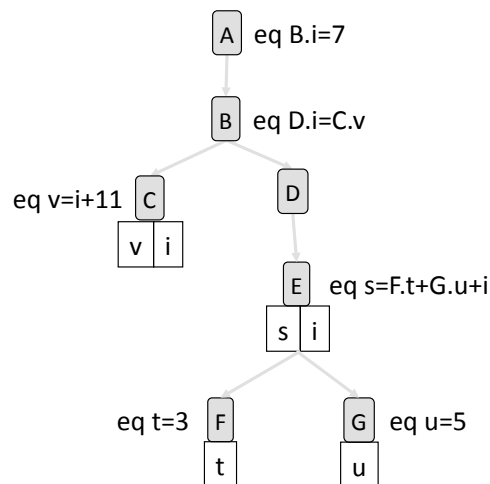
### 3 Basic attribution mechanisms

We will now look at the two basic mechanisms for defining properties of AST nodes: *synthesized* and *inherited attributes*. Loosely speaking, we may say that a synthesized attribute is used for propagating information upwards in the AST, whereas inherited attributes are used for propagating information downwards. In a compiler, type information is often propagated upwards in expression trees to perform type checking, and is typically represented by synthesized attributes. Information about declarations, on the other hand, is typically propagated from class or method nodes down to statements and expressions, using inherited attributes. These two basic kinds of attributes were introduced by Knuth in 1968 [Knu68]. The use of the term *inherited* is used here for historical reasons, and is different from and unrelated to the use of the term within object-orientation.

#### 3.1 Synthesized and inherited attributes

An attribute  $a$  is defined by an equation whose right-hand side is a function of other attributes, for example,  $a = f(b, c, d, \dots)$ . Attributes and equations are

declared in AST classes, so we can think of each AST node as having a set of declared attributes, and a set of equations. Some of the attributes are defined by equations in the node itself, so called *synthesized* attributes, and some are defined by equations in ancestor nodes, so called *inherited* attributes.



**Fig. 9.** The attributes E.s, F.t, G.u, and C.v are synthesized and have equations in the node they belong to. The attributes C.i and E.i are inherited. The equation in B applies only to the D subtree, and thereby to the E.i attribute. The equation in A applies to the C.i attribute, but not to the E.i attribute since it is shadowed by the equation in B.

Most attributes we introduce will be synthesized attributes. In the equation defining the attribute, we will use information in the node itself, say E, or by accessing its children, say, F and G. However, once in a while, we will find that the information we need is located in the context of the E node, i.e., in its parent, or further up in the AST. In these cases, we will introduce an inherited attribute in E, capturing this information. It is then the responsibility of all nodes that could have an E child, to provide an equation for that inherited attribute. The equation does not have to be in the immediate parent of E, but there must be an equation in some ancestor of E, on the way from the parent up to the root of the AST. If several of these nodes have an equation for the inherited attribute, the closest one to E will apply. See Fig. 9.

*Exercise 3.* What will be the values of the attributes in Fig. 9?

*Exercise 4.* An equation in node  $n$  for an inherited attribute  $i$  applies to the subtree of one of  $n$ 's children, say  $c$ . All the nodes in this subtree do not need

to actually have an  $i$  attribute, so the equation applies only to those nodes that actually have an  $i$  attribute. Which nodes in Fig. 9 are within the scope of an equation for  $i$ , but do not have an  $i$  attribute?

*Exercise 5.* In a correctly attributed AST, the attributes will have values so that all equations are fulfilled. How can the correct attribute values be computed? What different possibilities are there? (The actual way attributes are evaluated in JastAdd is discussed in Section 3.6.)

### 3.2 Reference and parameterized attributes

In JastAdd, synthesized and inherited attributes are generalized, as compared to the original formulation by Knuth. First, a JastAdd attribute is allowed to be a *reference* to an AST node, and in this way connect different AST nodes to each other, forming a graph. Furthermore, in an equation defining an attribute, it is allowed to use reference attributes and to access their local attributes. This allows information in one node to be propagated directly to a distant node in the AST, following the path of a reference attribute, rather than having to follow the tree structure. For example, if each use of an identifier has a reference attribute that points directly to the appropriate declaration node, information about the type can be propagated directly from the declaration to the use node.

A second generalization in JastAdd is that attributes may have parameters. A parameterized attribute will have an unbounded number of values, one for each possible combination of parameter values. For example, we may define an attribute `lookup(String)` whose values are references to declarations, different for each String value.

### 3.3 Thinking declaratively and object-oriented

In using attributes, with or without parameters, we can view them as methods of AST nodes. Attributes are similar to abstract methods, and equations are similar to method implementations. In fact, when accessing attributes, we will use Java method call syntax, e.g., `a()`, and when we write an equation, we can write the right-hand side as a Java method body. The similarity is particularly apparent for synthesized attributes, which are defined in the AST node itself, just like methods. Inherited attributes are different in that the attribute is declared in one node, but defined in another (in an ancestor).

Another difference between attributes and ordinary methods is that the equations defining attributes must not have any externally visible side effects. They may use local variables to compute some value, but may not change any global information, for example fields of AST nodes or global data. The reason for this is that they should not be viewed as a computation. Rather, we should think of the AST as having all the defined attributes, and that they have values such that the equations are fulfilled. It is not defined how many times an equation is evaluated. For efficiency, it might be evaluated just once, and the value stored for subsequent accesses. If the attribute is not accessed, the equation might

not be evaluated at all. Therefore, any externally visible side effects within the equations will not have a well-defined behavior.

When writing an attribute grammar, you should try to think declaratively, rather than to think about in which order things need to be computed. Think first what properties you would like the nodes to have, to solve a particular problem. In the case of type checking, it would be useful if each expression node had a `type` attribute. You then have to decide if it should be a synthesized or inherited attribute. If you need information from the node itself, or from its subtree, you should define it as a synthesized attribute. The `type` attribute should obviously be a synthesized attribute, since, for example, different kinds of expressions should have different `type` values. For example, an `add` node might need to use the `type` attributes of its children, i.e., its operands, to decide if its type should be `int` or `float`, or something else.

As you write the equations defining the attribute for different subclasses of expression, you can introduce inherited attributes if you find that you need information that is not available in the node or its subtree. For example, to define the `type` of an identifier expression, it would be good to have an attribute `decl` that points to the appropriate declaration node. This should be a synthesized attribute since it depends on the name of the identifier. However, it also depends on information outside of the identifier node, i.e., about what declarations there are in the program, so for this we need to introduce an inherited attribute. The next section will look at a similar example in more detail.

### 3.4 Example: Name analysis for state labels

In section 2 we discussed an attribute `target` for Transition objects, that should point to the appropriate target State object. This can be seen as a name analysis problem: We can view the states as declarations and the transitions as uses of those declarations. In addition to the `target` attribute we will define an analogous `source` attribute which points to the appropriate source State object. We start by declaring `target` and `source` as synthesized attributes of Transition. This definition would be easy if we had a parameterized attribute `State lookup(String label)` that would somehow find the appropriate State object for a certain label. Since we don't have enough information in Transition to define `lookup`, we make it an inherited attribute. By looking at the abstract grammar, we see that the StateMachine node can have children of type Transition (since Transition is a subclass of Declaration), so it is the responsibility of StateMachine to define `lookup`. (In this case, StateMachine will be the root of the AST, so there are no further ancestors to which the definition can be delegated.)

In StateMachine, we can define `lookup` simply by traversing the declarations, locating the appropriate state. To do this we will introduce a synthesized attribute `State localLookup(String label)` for Declarations. Fig. 10 shows the resulting grammar. We use a JastAdd aspect to introduce the attributes and equations using inter-type declarations.

There are a few things to note about the notation used:

```

aspect NameAnalysis {
    syn State Transition.source() = lookup(getSourceLabel()); // R1
    syn State Transition.target() = lookup(getTargetLabel()); // R2
    inh State Transition.lookup(String label); // R3

    eq StateMachine.getDeclaration(int i).lookup(String label) { // R4
        for (Declaration d : getDeclarationList()) {
            State match = d.localLookup(label);
            if (match != null) return match;
        }
        return null;
    }

    syn State Declaration.localLookup(String label) = null; // R5

    eq State.localLookup(String label) = // R6
        (label==getLabel()) ? this : null;
}

```

**Fig. 10.** An aspect binding each Transition to its source and target States.

**syn, inh, eq** The keywords **syn** and **inh** are used for indicating declarations of synthesized and inherited attributes. The keyword **eq** indicates an equation defining the value of an attribute.

**implicit equations** The declaration of a synthesized attribute may contain an implicit equation, as in R1, R2, and R5.

**equation syntax** Equations may be written either using value syntax as in R1, R2, R5, and R6:

```
attr = expr,
```

or using method syntax as in R4:

```
attr { ... return expr; }
```

In both cases full Java can be used to define the attribute value. However, there must be no external side-effects resulting from the execution of that Java code.

**equations for inherited attributes** R4 is an example of an equation defining an inherited attribute. The left-hand side of such an equation has the general form `Class.getChild().attr()`. This means that the equation is located in `Class`, i.e., as if inside a method in `Class`. It defines the attribute `attr` of all nodes in the subtree `getChild()`, which is a method in the traversal API. In R4, the child is a list, and the traversal method therefore has an argument: `int i`. In R4, the argument `i` is not used, because all the Declaration children should have the same value for `lookup`.

**default and overriding equations** Default equations can be supplied in superclasses and overridden in subclasses. R5 is an example of a default equation, applying to all Declaration nodes, unless overridden in a subclass. R6 is an example of overriding this equation for the State subclass.

*Exercise 6.* Consider the following state machine:

```
state S1;
state S2;
trans a: S1 -> S2;
```

Draw a picture similar to Fig. 9, but for this state machine, i.e., indicating the location of all attributes and equations, according to the grammar in Fig. 10. Draw also the values of the `source` and `target` attribute values. Verify that these values agree with the equations.

*Exercise 7.* In a well-formed state machine AST, all State objects should have unique labels. Define a boolean attribute `alreadyDeclared` for State objects, which is true if there is a preceding State object of the same name.

*Exercise 8.* In there are two states with the same name, the first one will have `alreadyDeclared=false`, whereas the second one will have `alreadyDeclared=true`. Define another boolean attribute `multiplyDeclared` which will be true for both state objects, but false for uniquely named state objects.

### 3.5 More advanced name analysis

The name analysis for the state machine language is extremely simple, since there is only one global name space for state labels. However, the principle, using parameterized inherited attributes like `lookup`, scales up to full programming languages. For example, to deal with block-structured scopes, the `lookup` attribute of a block can be defined to first look among the local declarations, and, if not found there, to delegate to the context, using the inherited `lookup` attribute of the block node itself. Similarly, object-oriented inheritance can be handled by delegating to a `lookup` attribute in the superclass. This general technique, using `lookup` attributes and delegation, is used in the implementation of the JastAddJ Java compiler. See [EH06] for details. That paper also covers the JastAdd implementation of type analysis for Java.

### 3.6 Attribute evaluation and caching

As mentioned earlier, the JastAdd user does not have to worry about in which order attributes are given values. The evaluation is carried out implicitly by the JastAdd system. Given a well-defined attribute grammar, once the AST is built, all equations will hold, i.e., each attribute will have the value given by the right-hand side of its defining equation. From a performance perspective, it is, however, useful to know how the evaluation is carried out.

The evaluation method is very simple. Each attribute is represented by a Java method, implemented using a method representing the equation right-hand side. When a value is accessed, its right-hand side method is simply called. For synthesized attributes, this is completely straight-forward. You may have noticed that synthesized attributes are very similar to ordinary methods, although

they should be free of side-effects. For inherited attributes, the implementation involves a little bit more administration, looking up the appropriate equation method in the parent, or further up in the AST.

Two additional issues are taken care of during evaluation. First, attribute values can be cached for efficiency. If the attribute is cached, its value is stored the first time it is accessed. Subsequent accesses will return the value directly, rather than calling the equation method. Attributes are explicitly declared to be cached by adding the modifier `lazy` to their declaration. Attributes that involve heavy computations and are accessed more than once (with the same arguments, if parameterized) are the best candidates for caching. For the example in Fig. 10 we could define `source` and `target` as cached if we expect them to be used more than once by an application:

```
...
syn lazy State Transition.source() = ...
syn lazy State Transition.target() = ...
...
```

The second issue is dealing with circularities. In a well-defined attribute grammar, ordinary attributes must not depend on themselves, directly or indirectly. If they do, the evaluation would end up in an endless recursion. Therefore, the evaluator keeps track of attributes under evaluation, and raises an exception at runtime if a circularity is found. Due to the use of reference attributes, there is no general algorithm for finding circularities by analyzing the attribute grammar statically [Boy05].

## 4 Composite attributes

It is often useful to work with composite attribute values like sets, lists, maps, etc. In JastAdd, these composed values are often sets of node references. An example is the `transitions` attribute of `State`, discussed in Section 2. It is possible to define composite attributes using normal synthesized and inherited attributes. However, often it is simpler to use *collection* attributes. Collection attributes allow the definition of a composite attribute to be spread out in several different places in an AST, each contributing to the complete composite value. Collection attributes can be used also for scalar values like integers and booleans, but using them for composite values, like sets, is more common. Before looking at collection attributes, we will take a look at how set-valued attributes can be defined using normal synthesized and inherited attributes.

### 4.1 Representing composite attributes by value objects

We will use objects to represent composite attribute values like sets. However, these objects should be regarded as *value objects*, i.e., once a value has been created, the object must never be changed. For the purpose of this tutorial, we will use a class called `PSet` to represent set values. `PSet` implements sets as a so

called persistent data structure, i.e., a data structure that has operations that can result in new values, but that do not mutate any existing value objects. For example, taking the union of two PSet objects will not change either of them. Fig. 11 shows the Java API for PSets.

```
class PSet<T> implements Iterable<T> {
    // Construction API
    public PSet(); // Constructor, returns an empty set of T objects.
    public PSet(T e); // Constructor, returns a singleton set containing e.
    public void add<T e>(); // Adds the element e to this object.
    public void add<PSet<T> s>(); // Adds all elements in s to this object.

    // Value API (non-mutating operations)
    public PSet<T> union(PSet<T> s); // Returns the union of this set and s.
    public PSet<T> union(T e); // Returns the union of this set and {e}
    public boolean contains(T e); // Returns true if this set contains e.
    public boolean isEmpty(); // Returns true if this is the empty set.
    public boolean equal(PSet<T> s); // Returns true if this set equals s.
}
```

**Fig. 11.** Java API for PSets (persistent sets)

When creating a new set value, the creation part of the API can be used, including the constructors and `add` operations that mutate the object under construction. After the creation is completed, it is only allowed to use the value API. Thus, within the right-hand side of an equation, a fresh PSet object can be created using the constructor, and mutated using `add`. But when accessing an attribute of an AST node, only the value API may be used.<sup>2</sup>

As an example of using PSets, Fig. 12 defines an attribute `states` in `StateMachine` that consists of references to all the `State` objects. Fig. 13 depicts the `StateMachine` of Fig. 3 with the `states` attribute.

Rule R1 defines the `states` set by traversing the declarations, and using a helper attribute `Declaration.states` that contains the set of states in that declaration. The helper attribute will be either a singleton set in case of a `State` (R3), or an empty set in case of a `Transition` (R2). Note that it is fine to use the mutating `add` operation in rule R1, since it changes `result` which is a value object under construction. Suppose we build two state machines `sm1`, and `sm2`. To construct the union of their states, we must not use the `add` method since it would corrupt existing value objects. Instead, we should use the `union` method in the value API:

```
sm1.states().union(sm2.states())
```

<sup>2</sup> The proper use of the construction and value APIs is currently not enforced by the JastAdd system.

```

aspect SetOfStates {
  syn PSet<State> StateMachine.Pstates() { // R1
    PSet<State> result = new PSet<State>();
    for (Declaration d : getDeclarationList()) {
      result.add(d.states());
    }
    return result;
  }
  syn PSet<State> Declaration.states() = new PSet<State>(); // R2
  eq State.states() = new PSet<State>(this); // R3
}

```

Fig. 12. Using PSets to define `states` as an ordinary (non-collection) attribute.

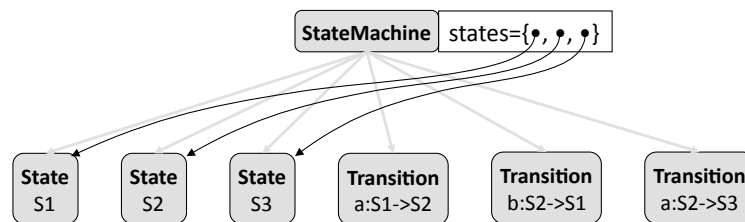


Fig. 13. The `states` attribute of the `StateMachine` is a set of references to the `State` objects.

While the definition of `states` in Fig. 12 is rather simple, there are many other cases when a traversal to find the appropriate objects may lead to a cumbersome specification, needing many helper attributes. A simpler solution might be possible using collection attributes.

## 4.2 Collection attributes

A *collection attribute* [Boy96,MEH09] has a composite value that is defined as a combination of *contributions*, instead of by an explicit equation. The contributions can be located anywhere in the AST. Instead of making it the responsibility of each collection attribute to find its contributions by traversing the AST, it is now the responsibility of each contributing node to declare its contribution to the appropriate collection attribute.

When computing the value of the collection attribute, each contributing object will mutate the partially built collection to add its contribution. Fig. 14 shows how to define the `states` attribute using JastAdd collections instead of ordinary attributes. As before, once the value is complete, i.e., when accessing the attribute via the AST, only the value API may be used.

```

aspect SetOfStatesUsingCollection {
  coll PSet<State> StateMachine.states() [new PSet<State>()] with add; // R1

  State contributes this // R2
  to StateMachine.states()
  for theStateMachine();

  inh StateMachine State.theStateMachine(); // R3
  eq StateMachine.getDeclaration(int i).theStateMachine() = this; // R4
}

```

Fig. 14. Defining `states` as a collection attribute.

Rule R1 declares the collection attribute `states`. It can be read as follows: There is a collection attribute of type `PSet<State>` for `StateMachine` objects, and it is called `states`. Its initial value (enclosed by square brackets) is `new PSet<State>()`, and contributions will be added with the method `add`.

The following syntax is used for declaring collection attributes:

```

collection-attribute-declaration ::=
  'coll' type nodeclass '.' attr '()'
  '[' initial-object ']'
  'with' contributing-method
  ['root' rootclass ]

```

We can note here that the *initial-object* should be a Java expression that creates a new object. The *contributing-method*, here `add`, is used by the underlying JastAdd machinery to add contributions, and should be a one-argument method that mutates the *initial-object*. It must be *commutative*, i.e., the order of calls should be irrelevant and result in the same final value of the collection attribute. Optionally, a *rootclass* can be supplied, limiting the contributions to occur in the AST subtree rooted at the closest *rootclass* object above or at the *nodeclass* object in the AST. If no *rootclass* is supplied, contributions can be located anywhere in the AST.

Rule R2 declares a contribution to the collection attribute. It can be read as follows: A `State` object contributes itself (`this`) to the attribute `states` in a `StateMachine` object. More precisely, (keyword `for`), the `StateMachine` object is `theStateMachine`.<sup>3</sup> Here, the attribute `theStateMachine` is a helper attribute, defined in rules R3 and R4.

<sup>3</sup> A more compact syntax would, in principle, be possible, eliminating the explicit mentioning of the `StateMachine` class. The reason for the current syntax is that the current version of JastAdd does not analyze the contents of Java expressions, such as `theStateMachine()`

The following syntax is used for declaring contributions:

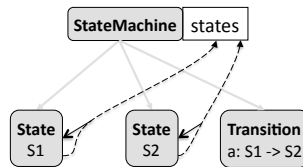
```

contribution-declaration ::=
  contributing-nodeclass 'contributes'
  ( expr [ 'when' cond ] , ',' )+
  'to' collection-nodeclass '.' attr '('
  'for' [ 'each' ] ref-expr

```

The *expr* should be a Java expression that has the type of the argument of the *contributing-method* in the collection declaration. In the example, there is an `add` method in `PSet<State>` which has the argument type `State`, so this condition is fulfilled. There can be one or more such contributions, separated by commas, and optionally they may be conditional, as specified in a `when` clause. The expression *ref-expr* should be a reference to a *collection-nodeclass* object. Optionally, the contribution can be added to a whole set of collection attributes by using the `each` keyword, in which case *ref-expr* should be a set of *collection-nodeclass* objects, or more precisely, it should be an object implementing Java's interface `Iterable`, and contain objects of type *collection-nodeclass*.

Fig. 15 depicts how the `State` objects contribute themselves to the `states` attribute in an example `StateMachine` object. The dashed arrows go from the contributing object to the collection attribute. The attached solid arrows show which objects are added to the collection.



**Fig. 15.** Dashed arrows show how contributing objects of class `State` add themselves (the short solid arrows) to the `states` collection attribute.

### 4.3 Defining transitions using a collection attribute

The `transitions` attribute discussed in Section 2 is a set attribute in `State` that contains references to all the `Transition` objects whose source is the `State` object in question. The knowledge of which objects to include in a `transitions` set is thus spread out in different `Transition` objects, making collection attributes a suitable design choice. Fig. 16 shows the definition of `transitions`. The contribution R2 is conditional since it should only be added when there is a correctly bound source state: if a transition refers to a non-existing source state, the `source` attribute will be null, and there is nowhere to add the contribution. Due to the `when` clause, there will be no contribution added in this case.

```

aspect TransitionsAttribute {
    coll PSet<Transition> State.transitions() [new PSet<Transition>()] with add; // R1

    Transition contributes this // R2
    when source() != null
    to State.transitions()
    for source();
}

```

**Fig. 16.** Defining `transitions` as a collection attribute.

*Exercise 9.* Make a drawing showing the values of the `transitions` attributes for the AST in Fig. 5. Use the same graphical notation as in Fig. 13.

*Exercise 10.* Make a drawing showing how objects are contributed to the `transitions` attributes for the AST in Fig. 5. Use the same graphical notation as in Fig. 15.

*Exercise 11.* Define a collection attribute `successors` for `States`. It should contain the set of `States` that are reachable from the state using exactly one transition step. Are there several ways of defining this attribute?

*Exercise 12.* Given the `successors` attribute, define a `predecessors` attribute for `State`, using a collection attribute. Hint: use the `for each` construct in the contribution.

*Exercise 13.* Collection attributes can be used not only for sets, but also for other composite types, like maps and bags, and also for scalar types like integers. Primitive types, like `int` and `boolean` in Java, need, however, to be wrapped in objects. Define a collection attribute that computes the number of transitions in a state machine.

*Exercise 14.* Define a collection attribute `errors` for `StateMachine`, to which different nodes in the AST can contribute objects describing static-semantic errors. The `errors` attribute could be a set of `ErrorMessage` objects, where `ErrorMessage` is an ordinary Java class. Give a suitable API for `ErrorMessage`. Transitions referring to missing source and target states are obvious errors. What other kinds of errors are there? Write a collection declaration and suitable contributions to define the value of `errors`.

For more examples of collection attributes, see the `Metrics` example, available at [jastadd.org](http://jastadd.org). This example implements Chidamber and Kemerer's metrics for object oriented programs [CK94]. The implementation is done as an extension to the `JastAddJ` compiler, and makes heavy use of collection attributes for computing the different metrics. Collection attributes are also used in the `Flow Analysis` example at [jastadd.org](http://jastadd.org), as described in [NNEHM08]. Here, `predecessors` in control-flow graphs, and `def` and `use` sets in dataflow, are defined using collection attributes.

#### 4.4 Evaluation of collection attributes

When accessing a collection attribute, JastAdd automatically computes its value, based on the existing contribution declarations. In general, this involves a complete traversal of the AST to find the contributions, unless the scope of the collection is restricted, using a `root` clause in the collection declaration. To improve performance, several collection attributes can be computed in the same traversal, either completely or partially. Given that a particular instance  $c_i$  of a collection attribute  $c$  is accessed, the default behavior of JastAdd is to partially compute all instances of  $c$ , so that further traversal of the AST is unnecessary when additional instances of  $c$  are accessed. The algorithm used is called *two-phase joint evaluation* [MEH09]. It is sometimes possible to achieve further performance improvements by using other algorithm variants. For example, the evaluation of several different collection attributes can be grouped, provided that they do not depend on each other. See [MEH09] for more details.

### 5 Circular attributes

Sometimes, the definition of a property is circular, depending ultimately on itself. When we write down a defining equation for the property, we find that we need the same property to appear at the right-hand side of the equation, or in equations for attributes used by the first equation. In this case, the equations cannot be solved by simple substitution, as for normal synthesized and inherited attributes, but a fixed-point iteration is needed. The variables of the equations are then initialized to some value, and assigned new values in an iterative process until a solution to the equation system is found, i.e., a fixed point.

The `reachable` attribute of `State` is an example of such a circularly defined property. In this section we will first look at how this property can be formulated and solved mathematically, and then how it can be programmed using JastAdd.

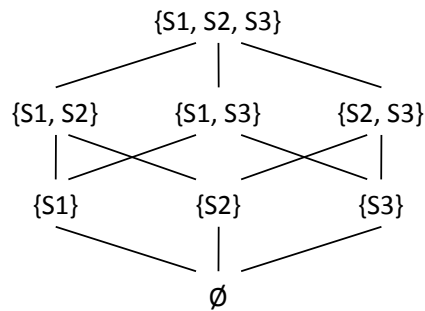
#### 5.1 Circularly defined properties

To define reachability for states mathematically, suppose first that the state machine contains  $n$  states,  $s_1..s_n$ . Let  $succ_k$  denote the set of states that can be reached from  $s_k$  through one transition. The set of reachable states for  $s_k$ , i.e., the set of states that can be reached via any number of transitions from  $s_k$  can then be expressed as follows:

$$reachable_k = succ_k \cup \bigcup_{s_j \in succ_k} reachable_j$$

We will have one such equation for each state  $s_k, 1 \leq k \leq n$ . If there is a cycle in the state machine, the equation system will be cyclic, i.e., there will be some *reachable* set that (transitively) depends on itself. We can compute a solution to the equation system using a least fixed point iteration. I.e., we use one *reachable* variable for each state, to which we initially assign the empty set. Then we

interpret the equations as assignments, and iterate these assignments until no *reachable* variable changes value. We have then found a solution to the equation system. The iteration is guaranteed to terminate if we can place all possible values on a lattice of finite height, and if all the assignments are monotonic, i.e., if they never decrease the value of any *reachable* variable.



**Fig. 17.** The sets of states for the state machine of Fig. 3 are arranged in a lattice.

In this case, the values are sets of states, and they can be arranged in a lattice with the empty set at the bottom and the set of all states in the state machine at the top. Fig. 17 shows the lattice for the state machine of Fig. 3. The lattice will be of finite height since the number of states in the state machine is finite. The assignments will be monotonic since the union operator can only lead to increasing values in the lattice. Because we start at the bottom (the empty set), we are furthermore guaranteed to find the *least* fixed point, i.e., the variables will stay at the lowest possible points in the lattice. If we have a cycle in the state machine, there may be additional uninteresting fixed points, for example by assigning the full set of states to *reachable* for all states on the cycle.

*Exercise 15.* For the state machine of Fig. 3, write down all the equations for *reachable*. Which are the variables of the equation system?

*Exercise 16.* What is the (least) solution to this equation system? Are there any more (uninteresting) solutions?

*Exercise 17.* Construct a state machine for which there is more than one solution to the equation system. What would be the least solution? What would be another (uninteresting) solution?

*Exercise 18.* Given the lattice of Fig. 17, give an example of a set operation that would not be allowed in the equations.

## 5.2 Circular attributes

In JastAdd, we can program circular properties like *reachable* by explicitly declaring the attribute as `circular`. and stating what initial value to use. The

attribute will then automatically be evaluated using fixed-point iteration. Fig. 18 shows the definition of the attribute `reachable` for `States`.

```

aspect ReachableAttribute{
  syn PSet<State> State.reachable() circular [new PSet<State>()]; // R1

  eq State.reachable() { // R2
    PSet<State> result = successors();
    for (State s : successors()) {
      result = result.union(s.reachable());
    }
    return result;
  }
}

```

**Fig. 18.** Defining `reachable` as a circular attribute.

Things to note:

- syntax** Synthesized, inherited and collection attributes can be declared as circular by adding the keyword `circular` after that attribute name. For synthesized and inherited attributes, an initial value also needs to be supplied, surrounded by square brackets as shown in the example above. For collection attributes, the initial object is used as the initial value.
- caching** Circular attributes are automatically cached, so adding the keyword `lazy` has no effect.
- equals method** The types used for circular attributes must have a Java `equals` method that tests for equality between two attribute values.
- value semantics** As usual, it is necessary to treat any accessed attributes as values, and to not change their contents. In the example, the local `result` variable in R2 is initially set to the value of the `successors()` attribute. It is therefore crucial that the non-mutating method `union` is used. Using the mutating method `add` instead could in this case change the values of existing attributes and lead to erroneous results. If we had instead initialized the `result` variable using a new fresh `PSet` object, we could have used the `add` method to change it locally within the rule.

*Exercise 19.* Make a drawing showing the values of the `reachable` attributes for the AST in Fig. 5. Use the same graphical notation as in Fig. 13.

*Exercise 20.* Make a drawing showing how objects are contributed to the reachable attributes for the AST in Fig. 5. Use the same graphical notation as in Fig. 15.

*Exercise 21.* Define `reachable` using a circular collection attribute. Hint: make use of the `predecessors` attribute defined in exercise 12.

*Exercise 22.* Set-valued circular attributes are common, but it is also possible to use other types whose values can be arranged in a lattice. One example is the simple boolean lattice with *false* at the bottom and *true* at the top, or the reverse lattice with *true* at the bottom and *false* at the top. Draw these two lattices. For each of the lattices, list which of the following operators are allowed: *and*, *or*, and *not*.

For more examples of JastAdd’s circular attributes, you may look at the the Flow Analysis example at [jastadd.org](http://jastadd.org) where intraprocedural control flow and dataflow is defined as an extension to JastAddJ, as described in [NNEHM08]. Here, the *in* set is defined as a circular attribute, and the *out* set as a circular collection attribute. In [MH07], there are examples of defining the properties *nullable*, *first*, and *follow* for nonterminals in context-free grammars, using JastAdd circular attributes. The *nullable* property is defined using a boolean circular attribute, and the two others as set-valued circular attributes. A variant of *follow* is defined in [MEH09] using circular collection attributes.

## 6 Conclusions

In this tutorial we have covered central attribution mechanisms in JastAdd, including synthesized, inherited, reference, parameterized, collection, and circular attributes. With these mechanisms you can address many advanced problems in compilers and other language tools. There are some additional mechanisms in JastAdd, not covered in this tutorial. One is *rewrites* [EH04], allowing sub ASTs to be replaced conditionally, depending on attribute values. This is useful when the AST constructed by the parser is not specific enough, or in order to normalize language constructs to make further compilation easier. Another mechanism is *nonterminal attributes* [VSK89] which allows the AST to be extended dynamically, defining new AST nodes using equations. This is useful for different kinds of macro expansion problems. In JastAddJ, nonterminal attributes are used for adding nodes representing instances of generic types. A future version of this tutorial might cover these mechanisms in greater detail.

The declarative construction of an object-oriented model is central when programming in JastAdd. The basic structure is always the abstract syntax tree (AST), but through the reference attributes, graphs can be superimposed. In this tutorial we have seen this through the addition of the **source** and **target** edges, and the **transitions** and **reachable** sets. Similar techniques are used to implement a compiler for a programming language like Java. Here, each use of an identifier can be linked to its declaration, each class declaration to its superclass declaration, and edges can be added to build control-flow and dataflow graphs. Once these graphs have been defined, further definitions are often made in terms of those graph structures rather than in terms of the tree structure of the AST.

An important design advice is to focus on thinking declaratively when programming in JastAdd. Think first about what attributes you would like the AST to have. Then, in defining these attributes, think of what other attributes that

would be useful, and that would make your equations simple. This will lead to the addition of new attributes. In this tutorial, we have mostly worked in the other direction, in order to present simple mechanisms before more complex ones. For a real situation, where you already know about the JastAdd mechanisms, you might have started out with the `reachable` attribute instead. In order to define it, it would have been useful to have the `transitions` attribute. To define the `transitions` attribute, you find that you need the `source` attribute, and so on.

As for normal object-oriented programming, naming is essential. Try to pick good descriptive names of both your AST classes and your attributes, so that the code you write is readable, and the APIs that the attributes produce will be simple and natural to use. For each attribute that you implement, you can write test cases that build up some example ASTs and test that the attributes get the intended values in different situations, so that you are confident that you have got your equations right.

JastAdd has been used for implementing both simple small languages and advanced programming languages. The implementation of our extensible Java compiler, JastAddJ, has been driving the development of JastAdd, and has motivated the introduction of many of the different mechanisms and made it possible to benchmark them on large programs [EH04,MH07,MEH09,NNEHM08]. Other advanced languages are being implemented as well, most notably an ongoing open-source implementation of the language Modelica which is used for describing physical models using differential equations [Mod09a,Mod09b,ÅEH08]. For more information about JastAdd, see [jastadd.org](http://jastadd.org).

**Acknowledgments.** The JastAdd system was jointly developed with my former PhD students Torbjörn Ekman and Eva Magnusson. For this tutorial I am particularly grateful to Torbjörn for joint work on constructing the state machine example. Thanks also to the students Nicolas Mora and Philip Nilsson whose comments helped me improve the readability of the tutorial.

## References

- [ÅEH08] J. Åkesson, T. Ekman, and G. Hedin. Development of a Modelica Compiler Using JastAdd. *Electr. Notes Theor. Comput. Sci.*, 203(2):117–131, 2008.
- [Boy96] John Tang Boyland. *Descriptive Composition of Compiler Components*. PhD thesis, University of California, Berkeley, September 1996. Available as technical report UCB//CSD-96-916.
- [Boy05] John Tang Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.
- [CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
- [EH04] Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, volume 3086 of *LNCS*, pages 144–169. Springer, 2004.

- [EH06] Torbjörn Ekman and Görel 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, 2006.
- [EH07a] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.
- [EH07b] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Proceedings of TOOLS Europe 2007, Journal of Object Technology*, 6(7), 2007.
- [Ekm06] Torbjörn Ekman. *Extensible Compiler Construction*. PhD thesis, Lund University, Sweden, June 2006.
- [Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the SIGPLAN symposium on Compiler Construction*, pages 85–98. ACM Press, 1986.
- [Hed00] Görel Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.
- [Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Correction: *Mathematical Systems Theory* 5, 1, pp. 95-96 (1971).
- [MEH09] Eva Magnusson, Torbjörn Ekman, and Görel Hedin. Demand-driven evaluation of collection attributes. *Automated Software Engineering*, 16(2):291–322, 2009.
- [MH07] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.
- [Mod09a] The Modelica Association, 2009. <http://www.modelica.org>.
- [Mod09b] Modelon AB. JModelica Home Page, 2009. <http://www.jmodelica.org>.
- [NNEHM08] E. Nilsson-Nyman, T. Ekman, G. Hedin, and E. Magnusson. Declarative intraprocedural flow analysis of Java source code. In *Proceedings of 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, 2008.
- [VSK89] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of PLDI '89*, pages 131–145. ACM Press, 1989.