Implementing Semantic Feedback in a Diagram Editor

Niklas Fors
Department of Computer Science
Lund University, Lund, Sweden
niklas@cs.lth.se

Görel Hedin
Department of Computer Science
Lund University, Lund, Sweden
gorel@cs.lth.se

ABSTRACT

In editors for visual languages it is often useful to provide interactive feedback that depends on the static semantics of the edited program. In this paper we demonstrate how such feedback can be implemented using reference attribute grammars. Because the implementation is declarative, it is easy to modularize compiler and editor computations, reusing the compiler's program model in the editor. Furthermore, the declarative approach makes it easy to keep the program model and view consistent during editing. The approach is illustrated using a function block diagram language, with visual feedback on, for example, type checking and cyclic data flow.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*; D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Visual languages, diagram, reference attribute grammars, jastadd

1. INTRODUCTION

In implementing language tooling, there is often the need for providing several related tools: textual editor, visual editor, compiler, program analyzers, etc. Developing such language tooling is typically very costly [22, 18], and there are many current efforts on providing meta-tooling to reduce these costs. Examples include both work based on metamodeling, like the Eclipse Modeling Framework (EMF), and work based on grammars, e.g., Spoofax [13].

GMLD '13, July 01 2013, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489827

While visual editors typically support structured editing, it is often useful to additionally provide *semantic feedback*, i.e., visualization and interaction that depends on static-semantic properties of the program. Examples include both visual display of static-semantic information, like displaying of type-checking errors, as well as interactive cues during editing gestures. An example could be to show if dragging a visual item to another location would change the semantics or not, or would introduce an error. If the underlying static-semantic analyses are done by a compiler, it is desirable to let the editor reuse this analysis, both in order to keep down development costs, and also to keep the tools consistent with each other.

We are currently exploring how reference attribute grammars (RAGs) [11] can be used in this context. RAGs are based on abstract syntax trees (ASTs), but provide reference attributes that link together AST nodes to form a graph. RAGs can also be integrated with EMF as shown by Bürger et al. [6], where containment relations correspond to the AST, and non-containment relations are mapped to reference attributes. RAGs have previously been shown useful for building extensible compilers for textual languages like Java [8] and Modelica [5].

We are applying RAGs for building tooling for a function block diagram language for control systems. The language, PicoDiagram, is based on existing languages in industrial use, and exhibits fairly advanced static-semantics, like diagram types, and data-flow based execution order. We have designed both a textual and visual syntax for PicoDiagram, and implemented both a compiler and a visual editor for it. The compiler is implemented in RAGs, using the metacompilation system JastAdd [12], and the editor is implemented using GEF, the Graphical Editing Framework in Eclipse.

In this paper, we investigate different opportunities for reusing static-semantics computations in the PicoDiagram compiler in order to provide semantic feedback during visual editing. In some cases, the compiler computations can be directly reused by the editor, and in other cases, they are modularly extended to provide specific feedback. This paper provides more details on the use of RAGs, compared to previous work [10].

The rest of this paper is organized as follows: Section 2 describes the PicoDiagram language, and Section 3 gives some background on RAGs. Sections 4, 5, and 6 describe different aspects of static-semantic analysis of PicoDiagram: name analysis, type checking, cyclic types, and cyclic connections. We explain how these aspects are implemented using RAGs, and how the implementation can be reused or

extended by the editor to provide semantic feedback. Section 7 describes related work, Section 8 provides a discussion and Section 9 concludes the paper.

2. PICODIAGRAM

PicoDiagram is an experimental language for control systems. It is inspired by a product from ABB, which in turn builds on the IEC-61131 standard for programmable logic controllers, including function block diagrams.

A PicoDiagram diagram consists of blocks and connections that model data flow. The blocks are instances of diagram types that may be defined either by other PicoDiagram diagrams, or by external C functions. We have designed PicoDiagram to have both a visual and a textual syntax. The textual syntax is a high-level model of a diagram, in the sense that visual properties such as shapes and colors are not part of the textual syntax. There are several benefits of having a textual syntax, for example, when generating diagrams or in concurrent development to allow the use of existing text-based merging tools.

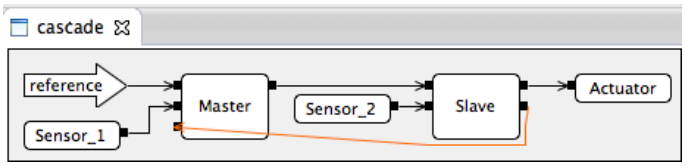
Figure 1 shows an example PicoDiagram program in both the visual and textual syntax. The program is a cascade regulator that controls an actuator using two controllers: a master controller uses input from one sensor to control the setpoint of the slave controller. The slave controller reads another sensor and sets the actuator value. There is also feedback information that goes from the slave to the master (the orange connection).

A diagram is executed periodically, reading/setting sensors and actuators with a given frequency. In each period, the blocks are executed in a sequence. In the case of a loop, like in the example, the compiler automatically breaks the loop by considering a certain connection as a *backwards connection*. In the visual editor, backward connections are colored orange. This is an example of giving the user semantic feedback. For values coming via backward connections, the old value from the previous period will be used.

The algorithm for computing the backwards connections makes use of a positional order between the blocks, see Section 6. For the visual syntax, the positional order corresponds to the distance to the origin, i.e., the upper left corner of the diagram. For the textual syntax, it corresponds to the declaration order. The execution order follows the forward data flow connections, and uses the positional order to define an unambiguous total order. In a previous paper, we defined the execution order declaratively on diagrams with acyclic data flow, and described how to provide semantic feedback on how blocks in a diagram can be moved without affecting the execution order [10]. In this paper, we extend this result by describing an algorithm to remove data flow cycles in diagrams using the positional order and give a more thorough description on the use of RAGs.

A diagram type can be instantiated several times. For example, a larger system may include several instances of the `Cascade` diagram.

We have implemented two different tools for PicoDiagram: a visual editor where the user can edit programs using the visual syntax, and a compiler that translates the text representation to C code. The visual editor stores the programs in the text representation, and it is also possible to use a normal text editor to edit the programs. Both tools reuse the same parser and core RAG specification, but they can also extend the RAG specification with their own attributes.




```
diagramtype Cascade(Int reference) {
  Sensor Sensor_1;
  Master Master;
  Sensor Sensor_2;
  Slave Slave;
  Actuator Actuator;
  connect(Sensor_1.value, Master.pv);
  connect(reference, Master.sv);
  connect(Master.u, Slave.sv);
  connect(Sensor_2.value, Slave.pv);
  connect(Slave.u, Actuator.value);
  connect(Slave.feedback, Master.slaveFeedback);
}
```

Figure 1: A cascade regulator in PicoDiagram. Visual and textual syntax.

For example, the editor can add attributes that aids the implementation of semantic feedback, and the compiler adds attributes for code generation. This way the tools are kept consistent with each other, yet can be tailored as desired.

3. REFERENCE ATTRIBUTE GRAMMARS

We use the semantic formalism *Reference Attribute Grammars* (RAGs) [11] to define the meaning of a diagram in the PicoDiagram language. A diagram is represented as an attributed *abstract syntax tree* (AST), typically created by a parser or by an interactive visual editor. The attributes are derived values computed given a set of equations and the AST. Equations are directed, with an attribute on the left hand side, and an expression over other attributes on the right hand side. The attributes are *declarative* in the sense that each attribute will have a value that is equal to the right-hand side of its defining equation, and the expressions inside an equation are forbidden to have any externally observable side effects. An *attribute evaluation engine* automatically evaluates the attributes, according to their dependencies.

RAGs extends Knuth's Attribute Grammars [14] with *reference* attributes, i.e., attributes whose values are references to other nodes in the AST. This makes is possible to superimpose graphs on top of the AST.

For implementation, we use the JastAdd metacompilation tool [12] which supports RAGs, object-oriented ASTs, and aspect-oriented modularization of RAG specifications. JastAdd also supports circular attributes [16], i.e., attributes that may transitively depend on themselves, and for which the evaluation engine uses fixed-point iteration to compute the value.

JastAdd attributes are accessed by calling them, like a method. In fact, the attributes are translated to Java methods by the JastAdd tool. For efficiency, the JastAdd attribute evaluator does not evaluate an attribute until its value is needed (i.e., until it is called), and evaluated attributes are cached for fast future access. The attribute caches are flushed whenever the AST is modified, for example after an edit action.

3.1 A Simple RAG Example

To illustrate how RAGs work, consider the following simple abstract grammar (written in JastAdd syntax):

```
A ::= B1:B B2:B;
B ::= <ID:String>;
C : B;
```

This abstract grammar corresponds to a Java class hierarchy, and defines that:

- `A`, `B`, and `C` are classes
- `A` has two children: `B1` and `B2`, both of type `B`
- `B` has a token: `ID` of type `String`
- `C` is a subclass of `B`

The JastAdd RAG specification below defines attributes over this abstract grammar. Line 1 declares an attribute `B.otherB` of type `B`. I.e., `otherB` is a reference attribute in `B` nodes, and that will point to another `B` node. Lines 2 and 3 show two different equations defining `otherB`: the first one holds for the `B1` child of an `A` node, and the other one for the `B2` child. The right-hand side of the equation is in the context of `A`. The attribute `otherB` is *inherited* (in the attribute grammar sense), meaning that the `B` node does not know how the value is defined, but delegates this responsibility to the parent node, i.e., to the `A` node in this case.[1]

```
1   inh B B.otherB();
2   eq A.getB1().otherB() = getB2();
3   eq A.getB2().otherB() = getB1();
```

The result of this specification is that the two `B` children refer to each other, thus forming a cyclic data structure, as shown for the example AST in Figure 2. This AST corresponds to the expression `new A(new B("B"), new C("C"))`. Note that one of the `B` nodes is actually of the subclass `C`, illustrating that attributes are inherited (in the usual object-oriented sense) by subclasses.

Because RAGs are declarative, the order of specification of individual attributes and equations is irrelevant, and they can be split up into different modules based on what is practical to reuse in different tools. The following small RAG module illustrates the modular addition of an attribute `B.name` of type `String`. This is a *synthesized* attribute, meaning that there must be an equation defining its value in the same node. Line 5 gives a definition of the attribute value for `B` nodes. Line 6 illustrates how this definition is overridden for `C` nodes, relying on the usual object-oriented features of inheritance and overriding. The equation on line 6 also illustrates how information in distant nodes can be accessed: the `ID` of the other `B` node is accessed via the reference attribute `otherB`.

```
4   syn String B.name();
5   eq B.name() = getID();
6   eq C.name() = otherB().getID();
```

[1]JastAdd also supports inheritance in the object-oriented sense, and if there is a risk on confusion, we will comment on what kind of inheritance we mean. Both uses of the term originate from the late 60s when both OO and AGs were invented.

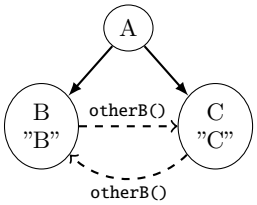


Figure 2: Example AST with reference attributes

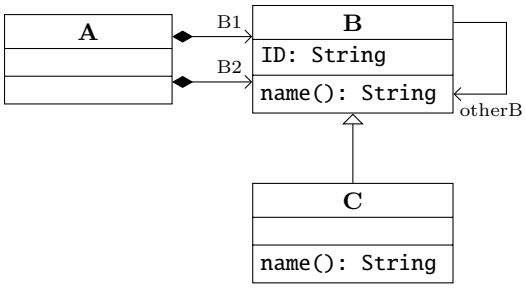


Figure 3: UML diagram for the RAG example

The RAG corresponds to the UML diagram depicted in Figure 3. The class hierarchy, the parent-child relations (i.e., containment relations), and UML attributes like `ID`, are specified in the abstract grammar. Non-containment relations like `otherB`, and methods like `name`, are specified as RAG attributes, and their values are computed automatically by the underlying RAG evaluation engine.

Additional JastAdd/RAG mechanisms used in this paper include parameterized attributes and circular attributes, and are explained as part of our discussion of the PicoDiagram compiler and editor.

4. NAME AND TYPE ANALYSIS

The name analaysis is concerned with binding name uses to name declarations. For PicoDiagram, we have two separate namespaces, one for types and one for variables. Type uses and type declarations are represented by the classes `TypeUse` and `TypeDecl`, respectively, and `VarUse` and `VarDecl` represent variables. The bindings are implemented by a reference attribute `decl` in `TypeUse` and `VarUse`, referring to the appropriate declaration node.

An example of an attributed AST is shown in Figure 4. The structure of this AST is defined by the core abstract grammar of PicoDiagram, which is shown in Figure 5.[2] In this grammar, we can see that a `Program` consists of a list of `DiagramType`s, which in turn consists of a name, `Parameter`s, `Block`s and `Connection`s. The classes `Parameter` and `Block` are subclasses to `VarDecl` and have a `TypeUse` each. The `Connection` class has at least one `VarUse`.[3]

Figure 6 illustrates a simple diagram type `S`, using the textual syntax. It has one input parameter `p`, a block `t` of type `T` and a connection between the local parameter `p` and the parameter `p2` in type `T`. There are three type uses in this example: the parameters `p` and `p2` use the primitive type `Int` and the block `t` uses the diagram type `T`. The variable uses

[2]The full PicoDiagram language contains additional constructs like structs, floats, booleans, etc.

[3]The source of a connection can be an integer constant.

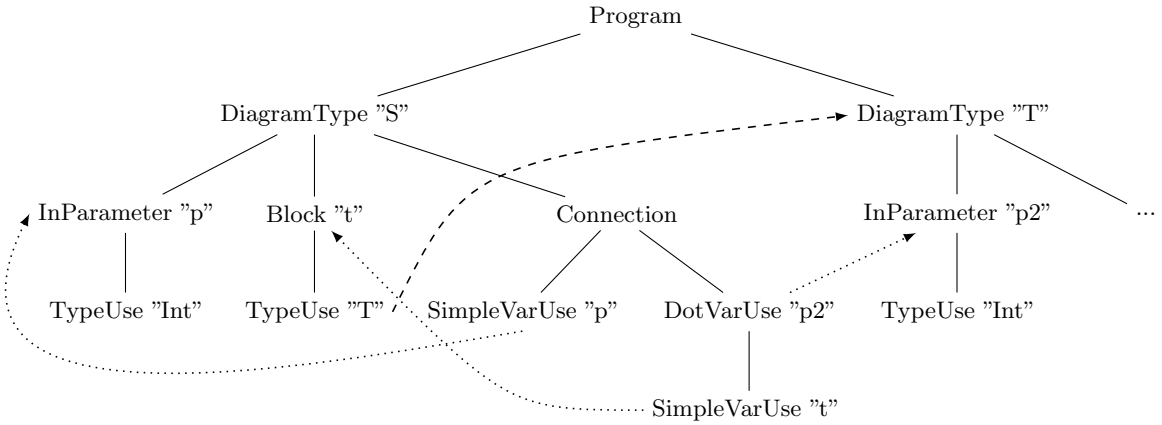


Figure 4: Example AST. Dashed lines are references from type uses to type declarations. Dotted lines are references from variable uses to variable declarations. The type `Int` and references to it have been omitted.

```
Program ::= DiagramType*;

abstract TypeDecl;
DiagramType : TypeDecl ::=
  <ID> Parameter* Block* Connection*;
IntType : TypeDecl;
BoolType : TypeDecl;
UnknownType : TypeDecl;

abstract VarDecl;
abstract Parameter : VarDecl ::= TypeUse <ID>;
InParameter : Parameter;
OutParameter : Parameter;
Block : VarDecl ::= TypeUse <ID>;

Connection ::= Source:Expr Target:VarUse;

abstract Expr;
IntConst : Expr ::= <Value:int>;
abstract VarUse : Expr;
SimpleVarUse : VarUse ::= <ID>;
DotVarUse : VarUse ::= VarUse <ID>;

TypeUse ::= <ID>;
```

Figure 5: Core abstract grammar for PicoDiagram. Abstract classes (like in Java) use the keyword `abstract`. Kleene star means a list of children.

```
diagramtype S(Int p =>) {
  T t;
  connect(p, t.p2);
}
diagramtype T(Int p2 =>) { ... }
```

Figure 6: Simple example for name analysis

```
syn TypeDecl TypeUse.decl()
  = lookupType(getID());

inh TypeDecl TypeUse.lookupType(String name);
eq Program.getDiagramType()
    .lookupType(String name) {
  TypeDecl typeDecl = lookupPredefType(name);
  if (typeDecl == null)
    typeDecl = lookupDiagramType(name);
  return typeDecl;
}
```

Figure 7: Name binding for types

are found in the connection: `p` and `t.p2`. Note that the binding of `p2` depends on the type of `t`. The corresponding AST of this example and the name bindings are shown in Figure 4.

As described earlier, both `TypeUse` and `VarUse` have a reference attribute `decl` that refers to the corresponding declaration. The RAGs implementation follows a typical pattern for name analysis where the `decl` attribute delegates the searching for the declaration to an inherited attribute `lookup` that is defined by the context in the AST [7]. The `lookup` attributes are examples of *parameterized* attributes, that can be thought of as functions, but where the results are cached (i.e., memoized).

Figure 7 shows the name binding for types, where the `decl` attribute uses the inherited attribute `lookupType` which in turn is defined by the `Program` node for its diagram types. The attribute first searches among the predeclared types (integer and boolean) using the attribute `lookupPredefType` and then continues among the user defined diagram types using the attribute `lookupDiagramType`. See the appendix on how these helper attributes are defined.

The implementation of name bindings for variables is shown in Figure 8. For `SimpleVarUse`, the attribute `decl` uses the `lookup` attribute directly. The enclosing diagram type defines the `lookup` attribute for all its children. It first searches among its parameters and then continues with searching among its blocks. The definitions of the helper attributes can be seen in the appendix. For `DotVarUse`, the attribute

```
syn VarDecl VarUse.decl();

eq SimpleVarUse.decl()
  = lookup(getID());

inh VarDecl VarUse.lookup(String name);
eq DiagramType.getChild().lookup(String name)
{
  VarDecl decl = localParameterLookup(name);
  if (decl == null)
    decl = localBlockLookup(name);
  return decl;
}


eq DotVarUse.decl()
  = getVarUse().decl() == null
  ? null
  : getVarUse().decl().findMember(getID());

syn VarDecl VarDecl.findMember(String name)
  = null;
eq Block.findMember(String name)
  = type().localParameterLookup(name);
```

Figure 8: Name binding for variables

```
syn TypeDecl TypeUse.type()
  = decl() != null
  ? decl()
  : program().unknownType();

syn TypeDecl VarDecl.type();
eq Parameter.type() = getTypeUse().type();
eq Block.type() = getTypeUse().type();

syn TypeDecl Expr.type();
eq VarUse.type()
  = decl() != null
  ? decl().type()
  : program().unknownType();
eq IntConst.type() = program().intType();
```

Figure 9: Type analysis

`decl` is defined by asking the declaration of its child if it has a member using the attribute `findMember`. For example, in the expression `t.p2`, we use the declaration of `t` to define `p2`. In this example, the declaration of `t` is a block, and `Block` defines `findMember` by searching among the parameters of its type, that is, `p2`.

Types of expressions and declarations are represented by `type` attributes that refer to `TypeDecl` AST nodes. They are straightforward to define with the help of the `decl` attributes, as shown in Figure 9. Predeclared types, like `intType`, are represented using attributes in the program root. Semantic feedback on errors, for example, connections that connect incompatible types, can be shown in the visual editor using, for instance, colors or icons and associated error messages.

5. CYCLIC DIAGRAM TYPES

If a diagram type contains a block of its own type (directly or transitively), this constitutes a compile-time error

```
diagramtype T() {
  S s;
}
diagramtype S() {
  T t;
}
```

Figure 10: Cyclic diagram types (a compile-time error)

```
syn Set<TypeDecl> TypeDecl.reachable()
  circular[new HashSet<TypeDecl>()];
eq TypeDecl.reachable()
  = new HashSet<TypeDecl>();
eq DiagramType.reachable() {
  Set<TypeDecl> set = new HashSet<TypeDecl>();
  for (Block b: getBlocks()) {
    if (b.type().isDiagramType()) {
      set.add(b.type());
      set.addAll(b.type().reachable());
    }
  }
  return set;
}

syn boolean DiagramType.isCircular()
  = reachable().contains(this);
```

Figure 11: Type reachability

since it would lead to endless unfolding of blocks. Figure 10 illustrates this error.

We use a circular attribute [16] to identify such cycles. The value of a circular attribute is computed using a fixed-point iteration. We define the circular attribute `reachable` that is the set of all reachable diagram types from a diagram type, as can be seen in Figure 11. If the diagram type is in this set, the diagram type is circular, otherwise it is not. The attribute is defined on `TypeDecl`, to avoid explicit type casts. The attribute returns an empty set for all other types than diagram types. For diagram types, it returns the types of its blocks and the reachable types from these types. The attribute `isCircular` on `DiagramType` tells whether the diagram type is circular or not, by using the attribute `reachable`.

5.1 Semantic Feedback

We can now provide semantic feedback for cyclic diagram types by coloring blocks red that are in a type cycle. For example, in viewing the diagram type `T` from the example in Figure 10, the component block `s` would be colored red as follows:

To aid the implementation of this visual semantic feedback, an attribute `isCircular` is defined for `Block` as follows:

```
syn boolean Block.isCircular() =
  type().reachable().contains(diagramType());
```

The attribute checks if the enclosing diagram type of the block is reachable from the type of the block. If this true, then the block is in a type cycle. The enclosing diagram type is accessed by the inherited attribute `diagramType`.

6. CYCLIC CONNECTIONS

The connections define data flow between blocks, transferring values from source blocks to target blocks. Normally, a connection's source block should therefore execute before the target block. However, this becomes problematic when the connections form a cycle, since we do not know which block to execute first. We solve this by constructing a graph where edges corresponding to certain connections are removed, so that the resulting graph becomes acyclic. All blocks are executed periodically, so for a connection corresponding to a removed edge, we use the source block value from the *previous period* and feed it into the target block.

The idea behind the algorithm for removing edges is to only remove those corresponding to connections that visually go backwards, meaning that the source block is further away from the origin than the target block. If there are several such connections in a cycle, we choose the one whose target is closest to the origin. This is simple to understand for a user of the visual editor, and fits with the way control diagrams are usually layed out.

We model the diagram as a directed graph. In the diagram, the connections go between specific *ports* on the blocks, representing different variables. For the constructed graph, we abstract this to edges between the blocks, since we are only interested in ordering complete blocks. So if there are more than two connections between two specific blocks, they will be represented by a single edge in the graph.

The algorithm for breaking cycles is shown in Figure 12 and we use strongly connected components (SCCs) to detect cycles. An SCC is the maximal set of vertices where there is a path from all vertices in the SCC to all other vertices in the SCC, and can be computed in linear time using Tarjan's algorithm [21]. The cycle-breaking algorithm is iterative and removes edges until the graph is acyclic. For each iteration, all SCCs are computed. For each SCC consisting of more than one vertex, we select the vertex v_1 that is closest to the origin in the SCC. We then remove all edges from all vertices in the same SCC to v_1. The function $po(v)$ maps a block to an integer and preserves the positional order.

An example of a cyclic graph is shown in Figure 13. Running the algorithm on this graph, two SCCs are computed during the first iteration: $\{\alpha\}$ and $\{a, b, c, d, e\}$. We are only interested in SCCs with more than one vertex, that is, the latter SCC. In this SCC, vertex a is closest to the origin. We thus remove all edges from vertices in the same SCC to a, that is, the edge (c, a). During the second iteration, we compute the SCC $\{c, e\}$ and therefore remove the edge (e, c). The graph is now acyclic and we are finished.

The algorithm is implemented as a method `breakCycles` on `DiagramType` and uses attributes to obtain the information needed to construct the original cyclic graph. The method returns a wrapper object consisting of all predecessor and successor sets for the corresponding acyclic graph. The attribute `DiagramType.predSucc` is defined by calling `breakCycles`, as can be seen in Figure 14. This is fine since although `breakCycles` is implemented imperatively, it does not result in any externally observable side-effects. Note that `breakCycles` is only computed once for each diagram

remove all edges (v, v)
while graph has cycles **do**
  compute all SSCs
  for all SCC where $|SCC| > 1$ **do**
    let $v_1 \in SCC$, where $\forall v \in SCC : po(v_1) \leq po(v)$
    remove all edges (v, v_1), where $v \in SCC$
  end for
end while

Figure 12: Algorithm for breaking cycles

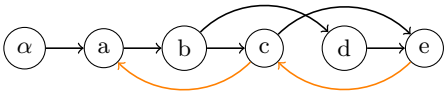


Figure 13: Cyclic graph. Orange edges are removed.

type, since JastAdd caches attribute values. The wrapper object consists of two maps: `pred` maps a `Block` to its predecessors and `succ` maps it to its successors. We use these two maps to define the attributes `pred()` and `succ()` on `Block`.

6.1 Semantic Feedback

Connections corresponding to a removed edge will have a different meaning than normal edges, since their values will be delayed one period. To help the user in understanding this, we provide semantic feedback by coloring these connections orange, as can be seen in Figure 1. To aid this implementation, an attribute `isBroken` is defined on `Connection`. A connection is defined as broken if both its source and targets are blocks, and if the target block is not a successor to the source block in the acyclic graph, see Figure 15.

We also provide interactive semantic feedback while the user moves a block, since this could affect how cycles are broken, see Figure 16. Here, the user is dragging the `Slave` block, and the green box shows its current position. The

```
syn PredSucc DiagramType.predSucc()
  = breakCycles();

inh Set<Block> Block.pred();
eq DiagramType.getBlock(int i).pred()
  = predSucc().pred.get(getBlock(i));

inh Set<Block> Block.succ();
eq DiagramType.getBlock(int i).succ()
  = predSucc().succ.get(getBlock(i));
```

Figure 14: Attributes `predSucc()`, `pred()` and `succ()`

```
syn boolean Connection.isBroken() {
  if (getSource().isBlock()
      && getTarget().isBlock()) {
    Block src = getSource().block();
    return !src.succ().contains(getTarget());
  } else {
    return false;
  }
}
```

Figure 15: Attribute `isBroken`

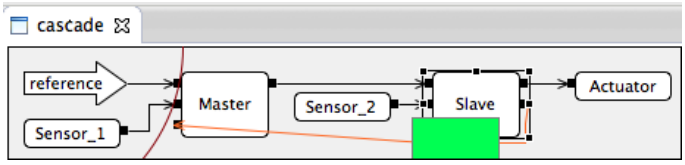


Figure 16: Semantic feedback for the cascade regulator. Moving the `Slave` block to the left of the red circle segment will change how the cycle is broken.

block stays green as long as releasing the block would not change the cycle breaking. However, if the user drags the block to the far left, across the red circle segment, i.e., closer to the origin than the `Master` block, the box will turn yellow, indicating that releasing it at that point may change the way cycles are broken.

In the general case, there could be both an upper and a lower bound for how a block could be moved without changing the cycle breaking. For example, how can we move the block c in Figure 13 without changing how the cycles are broken? By looking at the algorithm in Figure 12, we can see that only edges between vertices in the same SCC are removed, meaning that c is only constrained by vertices in the same SCCs. Block c is in two SCCs: $\{a, b, c, d, e\}$ for iteration 1 and $\{c, e\}$ for iteration 2. For each SCC, the only way to change which edges that are removed is to change the first vertex v_1 to another vertex. For block c, we can do this by moving c before a (first SCC) or after e (second SCC). Hence, c is constrained by a as the lower bound and e as the upper bound. The general constraint for all SCCs is $\forall v \in SCC : po(v_1) \leq po(v)$.

To implement the move feedback, we changed the implementation of the method `breakCycles` to also return information about what vertices a block is constrained by. We then use this information in the visual editor to provide the semantic feedback.

7. RELATED WORK

The focus of this paper is on reusing static semantics between different tools and providing semantic feedback in the visual editor.

In the Eclipse world, there are several ways to create a visual editor, such as using the Graphical Editing Framework (GEF) [1], the Graphical Modeling Framework (GMF) [2] or Graphiti [2]. The visual editor for PicoDiagram has been created using GEF. GMF is built on top of GEF and requires an EMF model to define a visual editor. These projects focus on lowering the burden for creating visual editors, and they focus less on semantic analysis and semantic feedback. One interesting work by Bürger et al. [6] combines RAGs and EMF, to define the static semantics of EMF models using RAGs. By building on their work, one possible future direction would be to use GMF instead of GEF to create the visual editor for PicoDiagram, in order to decrease the effort spent on the visual editor.

The blocks in PicoDiagram are instantiations of diagram types, and the appearance of these blocks depends on their types. A diagram type defines how many parameters it has and the visual representation of a block of this type reflects this. The language workbench MetaEdit+ [3] supports, since version 5.0, something they call dynamic ports, which they have created to support situations like this.

Another tool is xText [9] that is used for creating textual domain specific languages. It supports name binding for languages with simple name rules, otherwise the user needs to provide a Java implementation of the name binding. Also in the Spoofax language workbench, which has been used for graphical languages [23], a DSL is developed for name binding [15]. While such specific support can be useful in many cases, it is limited, both in what domain is supported (for example, name analysis), and in the generality of how that domain is supported (for example, limited to certain known kinds of scope rules). In contrast, RAGs is a general declarative formalism for defining *any* static-semantic analysis, and has been successfully used to implement complex name and type analyses for languages like Java [8] and Modelica [5].

Many metamodelling-based tools, including EMF, MetaEdit+, and xText, use constraint languages like OCL (Object Constraint Language) for specifying static-semantic constraints on the model. RAGs serve a similar purpose, but in addition supports *building* derived parts of the model itself, through the use of the reference attributes. Some of these derived parts, e.g., name bindings, may correspond to explicitly defined non-containment relations in EMF, whereas others go beyond EMF+OCL, like the representation of the broken cycles in section 6.

Schmidt et. al. [19] have used attribute grammars to implement visual editors. They use something called visual patterns that are predefined reusable implementations of common visual representations, such as lists, graphs, tables and line connections. These patterns have been implemented using attribute grammars. The user can refine these patterns and also create new visual patterns. The focus of their work is on using attribute grammars to define visual languages. A difference is that we use reference attributes and they do not. It would, however, be interesting to combine visual patterns with our approach.

In this paper, we handle cyclic data flows by removing connections based on the layout. In Simulink, a diagram can contain loops, which are called algebraic loops, that Simulink tries to solve mathematically [17]. If a loop contains a block with a discrete value as output, then Simulink cannot solve that loop. When Simulink cannot solve an algebraic loop, the user can add a so-called delay block to explicitly break the loop. In PicoDiagram, when a connection is removed due to a cycle, a delay is added implicitly, which corresponds to a delay block in Simulink.

Modelica is a language for modeling and simulating physical systems [4]. It has also a textual and a visual syntax, like PicoDiagram. However, in contrast to PicoDiagram, its semantics is not layout-dependent.

8. DISCUSSION

The declarative nature of attribute grammars allows attributes to be reused by different tools, for example, by the visual editor to provide semantic feedback. The attributes are declarative in the sense that they always compute the same value and no externally visible side-effects are allowed. This means that we can reuse the attributes in the visual editor without thinking about in what order they are computed. By reusing attributes between different tools, it is easier to keep the tools consistent with each other and the semantics is only required to be specified once.

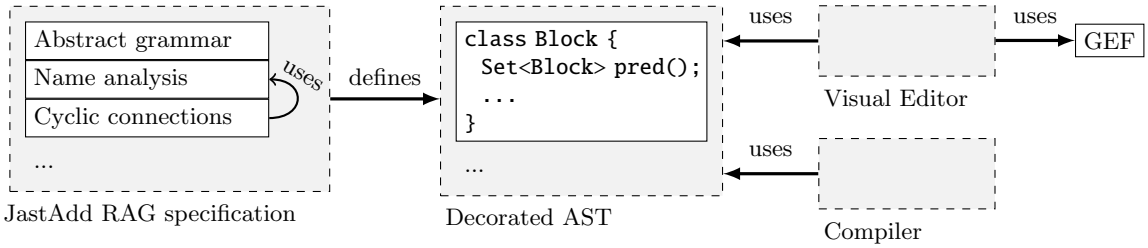


Figure 17: The visual editor and the compiler reuse the same decorated AST.

Figure 17 shows the overall architecture of how the RAG specification defines the attributed AST which is reused by both the compiler and the editor. In addition to this, the compiler and the visual editor can modularly add new attributes.

Attributes are computed values given a set of equations and an AST. When the user changes something in the visual editor, for example, adding a connection or a block, attribute values may become invalid, since the AST has changed. When the AST changes, we currently flush all attribute values, and they are recomputed again when they are needed (due to on-demand evaluation). In industry, large problems are modularized to many smaller diagrams, often limited to at most 50 blocks, since larger diagrams are difficult to understand and to fit on screen or paper. For PicoDiagram, there are no performance problems with individual diagrams: feedback is immediate even when interacting with diagrams with 500 blocks and 500 connections. However, further studies are needed to see how our approach scales to large libraries of diagrams. If performance needs to be improved, a possibility is to apply incremental evaluation for RAGs, where flushing is limited to attributes that depend on the AST change. Such incremental evaluation is currently investigated by Söderberg et. al. [20].

The base structure in EMF is a graph, whereas for RAGs it is a tree (the AST). With reference attributes, a graph can be super-imposed on the tree. We think having a high-level textual syntax also for a visual language has huge benefits. It makes the stored programs independent of tooling, and allows all existing text-level tools to be used. At the same time, the trees created by the parser are automatically decorated with attributes representing graphs, making the representation suitable for visual editors. However, the visual editor cannot change the reference attributes directly, but needs to transform visual editing operations to corresponding tree changes.

9. CONCLUSION

In this paper we have shown how to implement semantic feedback in a visual editor by reusing the static semantic specification. To only specify the static semantics once, and reuse the attributed AST in different tools, makes it is easy to keep the tools consistent with each other. As a case study, we have created a language called PicoDiagram that is similar to function block diagrams. The language has both a textual and a visual syntax, and we define its semantics using reference attribute grammars (RAGs), which is a declarative, but executable, formalism. For PicoDiagram, we have created a compiler and a visual editor that both reuse the same semantic specification including name analysis, type analysis, detection of cyclic diagram types and removal of cyclic connections based on layout. Then we showed examples of semantic feedback in the visual editor, for example, coloring blocks red that form a cyclic diagram type and displaying how a block can be moved without changing how the cyclic connections are broken.

In the future we would like to generate the visual editor from a specification, instead of coding it manually using GEF. It would be interesting to combine RAGs with EMF and GMF to create a visual editor, by building on the work by Bürger et al. [6]. It would also be interesting to use incremental evaluation [20] in the visual editor.

10. ACKNOWLEDGEMENTS

We would like to thank Ulf Hagberg, Christina Persson and Stefan Sällberg at ABB for sharing their expertise about the ABB tools for building control systems. This work was partly financed by the Swedish Research Council under grant 621-2012-4727.

11. REFERENCES

[1] The graphical editing framework (gef). `http://www.eclipse.org/gef/`.

[2] The graphical modeling project. `http://www.eclipse.org/modeling/gmp/`.

[3] Metaedit+ 5.0. `http://www.metacase.com/`.

[4] The modelica association. `http://www.modelica.org`, 2013.

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

[6] C. Bürger, S. Karol, C. Wende, and U. Aßmann. Reference attribute grammars for metamodel semantics. In *Software Language Engineering (SLE 2010)*, pages 22–41, 2011.

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

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

[9] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309. ACM, 2010.

[10] N. Fors and G. Hedin. Reusing semantics in visual editors: A case for reference attribute grammars. In

Proc. 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013), volume 58. ECEASST, 2013.

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

[12] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Sci. of Comp. Prog.*, 47(1):37–58, 2003.

[13] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA 2010*, pages 444–463. ACM, 2010.

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

[15] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering, 5th International Conference, SLE 2012*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2013.

[16] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[17] MathWorks. *Simulink documentation, Simulating Dynamic Systems*, r2013a edition.

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

[19] C. Schmidt and U. Kastens. Implementation of visual languages using pattern-based specifications. *Softw., Pract. Exper.*, 33(15):1471–1505, 2003.

[20] E. Söderberg and G. Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Lund University, April 2012. LU-CS-TR:2012-249, ISSN 1404-1200.

[21] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[22] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[23] O. van Rest, G. Wachsmuth, J. Steel, J. G. Süß, and E. Visser. Robust real-time synchronization between textual and graphical editors. In *International Conference on Model Transformation (ICMT 2013)*, Budapest, Hungary, June 2013.

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

APPENDIX

The attributes `lookupPredefType` and `lookupDiagramType` that are used in the name analysis for type names are shown in Figure 18. The attribute `lookupPredefType` access the predefined types using the non-terminal attribute (NTA) [24] `predefTypeDecls`. The value of an NTA is a subtree and is defined by an equation. The equation for the attribute `predefTypeDecls` defines a list of `TypeDecl`s corresponding to predefined declarations such as the integer and boolean type. The attributes `localParameterLookup` and `localBlockLookup` that are used in the name analysis for variable names are shown in Figure 19. These attributes access their children to find the corresponding declaration.

```
syn TypeDecl Program
    .lookupPredefType(String name) {
  for (TypeDecl td: predefTypeDecls())
    if (td.name().equals(name))
      return td;
  return null;
}
syn TypeDecl Program
    .lookupDiagramType(String name) {
  for (DiagramType dt: getDiagramTypes())
    if (dt.getID().equals(name))
      return dt;
  return null;
}
```

Figure 18: Helper attributes for looking up type names

```
syn VarDecl TypeDecl
  .localParameterLookup(String name) = null;
eq DiagramType
    .localParameterLookup(String name) {
  for (Parameter p: getParameters())
    if (p.getID().equals(name))
      return p;
  return null;
}

syn Block DiagramType
    .localBlockLookup(String name) {
  for (Block b: getBlocks())
    if (b.getID().equals(name))
      return b;
  return null;
}
```

Figure 19: Helper attributes for looking up variable names