Electronic Communications of the EASST
Volume 58 (2013)

Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013)

Reusing Semantics in Visual Editors: A Case for Reference Attribute Grammars

Niklas Fors and Görel Hedin

13 pages

Guest Editors: Matthias Tichy, Leila Ribeiro
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Reusing Semantics in Visual Editors: A Case for Reference Attribute Grammars

Niklas Fors and Görel Hedin

Department of Computer Science
Lund University, Lund, Sweden
(niklas.fors|gorel.hedin)@cs.lth.se

Abstract: The semantic formalism reference attribute grammars (RAGs) allows graphs to be superimposed on abstract syntax trees. This paper investigates how RAGs can be used to model visual languages, with a case study of a control language that also has a textual syntax. The language contains blocks on which a total execution order is defined based on connections and layout information. One strength of RAGs is reusability, and we demonstrate this by reusing the definition of the execution order in the visual editor to provide semantic feedback to the user.

Keywords: reference attribute grammars, JastAdd, visual languages

1 Introduction

Domain-specific languages (DSLs) are used in many engineering areas to support concise and natural descriptions of models, using both textual and visual languages, see e.g., [Wil04].

Tool development for languages, like compilers and editors, is typically very costly [DKV00, MHS05], and numerous reusable frameworks and generative techniques have been developed to help bring down costs. Recent examples include model-based metaprogramming systems such as the ecosystem around the Eclipse Modelling Framework (EMF), as well as generators based on formal specifications like term rewrite rules and attribute grammars (AGs), for example, Spoofax [KV10], and JastAdd [HM03].

While AGs decorate abstract syntax trees (ASTs) with attributes [Knu68], JastAdd is based on *reference AGs* (RAGs) [Hed00]. In RAGs, attributes can be *references* to other AST nodes, thereby superimposing graphs on the AST. JastAdd has been successfully used for building reusable compilers for textual languages like Java and the textual syntax of Modelica [EH07, ÅEH10], and we are now investigating their use for supporting visual languages and editors. Since visual languages are typically more natural to model using graphs than trees, we expect RAGs to be useful also for such languages. More specifically, our goals include:

- defining abstract models that can have both textual and visual syntax
- defining semantic analyses on the common model, for reuse in both compilers and editors
- using semantic analyses to obtain advanced language-based support in visual editors

As a case study, we have used JastAdd to develop a visual editor and a compiler for a simple but non-trivial visual language, *PicoDiagram*, demonstrating how the same model and analyses can be reused for both tools. PicoDiagram is inspired by a product from ABB, which in turn is

based on the IEC 61131-3 standard for programmable logic controllers. The language is (perhaps) unusual in that the visual layout of the elements in a program is used for disambiguating the execution order. This paper has the following contributions:

- The design of a textual syntax that allows the layout-specific semantics of PicoDiagram to be defined without using low-level graphical coordinates. This makes it possible to use the same abstract RAG model for both the textual and visual syntax. (Section 3.2)
- A high-level mathematical formulation of the layout-specific semantics, and its straightforward declarative implementation using RAGs. (Section 4)
- A demonstration of how the semantics can be reused to support advanced features in the visual editor. For example, we reuse the specification of the execution order to provide semantic feedback on how a block can be moved without changing this order. (Section 5)

Background on JastAdd is given in Section 2, and PicoDiagram is described in Section 3. Section 4 explains the declarative computation of the execution order, and Section 5 discusses how this computation is reused by the visual editor. Section 6 evaluates the approach, Section 7 discusses related work, and Section 8 concludes and discusses future work.

2 JastAdd

The semantic specification of the control language is defined declaratively using the metacompilation system JastAdd [HM03] which is based on object-oriented concepts, reference attribute grammars (RAGs) [Hed00], and inter-type declarations [KHH+01]. Traditional Knuth-style attribute grammars [Knu68] support *synthesized* and *inherited* attributes, used for propagating information downwards and upwards in the AST, respectively. RAGs generalize Knuth's AGs by supporting *reference* attributes, i.e., attributes that refer to other AST nodes, and *parameterized* attributes that provide one value per combination of parameter values, similar to a function.

A RAG decorates an AST with attributes, thereby describing the meaning of the AST. The attribute values are defined declaratively using *equations* and a generated attribute evaluator uses the equations to compute the value of any attribute in any AST. For instance, a name analysis RAG can provide equations defining reference attributes that connect use nodes to their corresponding declaration node. For further examples of uses of attributes, and additional kinds of attributes, see, e.g., [Hed09].

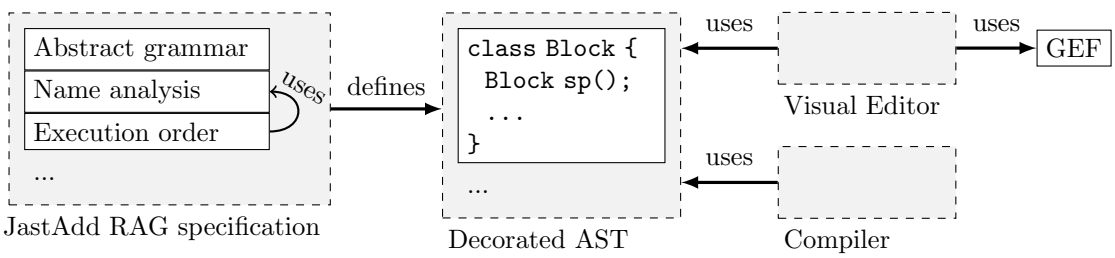


Figure 1: The visual editor and the compiler reuse the same decorated AST.

JastAdd represents an AST with Java objects, which are instances of classes that are generated from a user-defined *abstract grammar*. JastAdd does not care how the AST is created; it can be created by a third-party parser or by writing Java code directly. For each attribute, JastAdd generates a Java method, enabling attributes to be invoked from ordinary Java programs. The JastAdd attribute evaluator is *demand-driven* so that an attribute is not evaluated until it is invoked. Other attributes needed for its evaluation are then also evaluated, and caching is used to speed up multiple accesses of the same attribute. If the AST is changed (by an editor), invalid caches are flushed. This way, programs like a compiler or a visual editor can treat the AST as always being completely evaluated, and can reuse the same decorated AST, as Figure 1 shows.

The PicoDiagram editor is built on the graphical editing framework GEF, which is based on Eclipse and the model-view-controller (MVC) pattern. We use the AST as the model, and all changes in the editor are first made in the AST and then propagated back to the view using the Observer pattern.

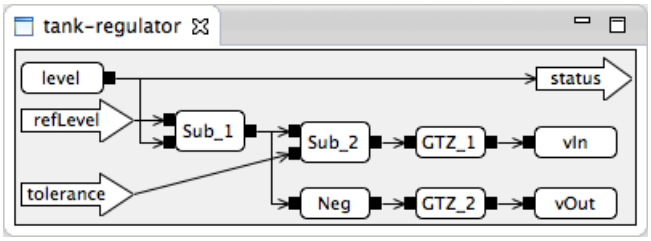
3 PicoDiagram—A Visual Control Language

Many control systems are built using programmable logic controllers (PLCs). A common way of programming them is to use visual *function block diagrams* (FBDs) like those defined in the IEC 61131-3 standard. An FBD consists of blocks and connections between them that describe the data flow. The blocks in a diagram are typically executed in periodic *scans*, based on sampling. Different FBD languages use different ways of defining the execution order. The PicoDiagram language is a prototype language inspired by an FBD language developed by ABB. Here, the connections define a partial execution order, and to obtain a total order, the layout of the blocks is used as well: if two blocks are unordered with respect to connections, the graphical positions of these and other blocks will influence the total order. This allows an engineer to easily understand the execution order just by looking at the diagram. The total order can be important for the semantics as blocks may use shared variables. If a diagram contains feedback loops, the data flow will be delayed to the next scan at certain connections, effectively breaking the loops. For the purpose of this paper, we therefore assume that each FBD is acyclic.

3.1 Tank Regulator Example

As an example, consider an on-off regulator for a tank containing liquid. The regulator uses two valves, one for filling (`vIn`) and one for emptying (`vOut`), to adjust the current liquid level (`level`) to a reference level (`refLevel`). The valves can be either open or closed. To avoid oscillation, the filling valve accepts a tolerance value (`tolerance`); the filling valve is only open if the error is larger than the tolerance value. The regulator model can be seen in Figure 2. Note that `Sub_1` produces data for two subsequent blocks, and it is the visual placement of the blocks that determines which one to execute first. In this simple case, `Sub_2` should execute before `Neg`, since `Sub_2` is closer to the origin (top left corner) than `Neg`, and there are no other edges that influence the order.

The execution order of the regulator model is shown in Figure 3a. Figure 3b shows how the execution order is changed when the grey block is moved in the editor. Note that the execution



(a) Visual syntax

```
diagramtype tank-regulator(
    Int refLevel, Int tolerance
    => StatusData status) {
  StatusMonitor level;
  Sub Sub_1;
  ...
  connect(refLevel, Sub_1.in1);
  ...
}
```

(b) Textual syntax

Figure 2: A regulator model for a tank with two input parameters (`refLevel` and `tolerance`) and one output parameter (`status`). White rounded rectangles represent blocks. `Sub` means subtraction, `Neg` negation, and `GTZ` means greater than zero and returns a boolean value. A valve takes a boolean value, indicating if it should be open or not. The data flow is from left to right.

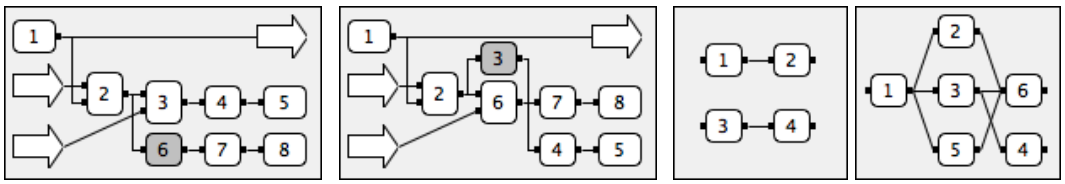
order is changed also for the succeeding blocks, following the intuition that a chain of connected blocks is executed in sequence.

3.2 Textual Syntax and Declaration Order

FBD languages are visual, and although the programs can be serialized and stored in files, this is typically done in a low-level format not primarily intended for viewing or editing, but that may contain generated redundant information, and mix graphical properties like shape, color and coordinates with constructs of semantic importance.

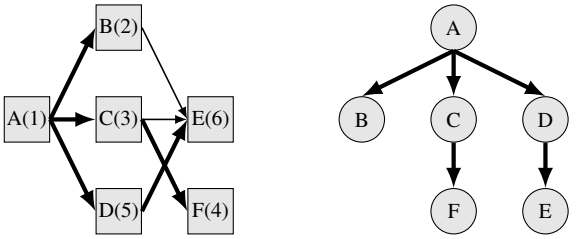
In contrast, for PicoDiagram, we have defined both a visual and a textual syntax, see Figure 2. The textual syntax is high-level and readable, and is useful, for example, for generating and merging diagrams. Even if graphical merging is desirable, textual syntax enables us to use existing merging techniques and tools without any effort.

However, since the execution order depends on the layout, we would like to design the textual syntax in such a way that the same execution order can be derived as from the visual diagram, but without using explicit block coordinates. We do this by defining a total *positional order* on



(a) (b) (c) (d)

Figure 3: Examples of different execution orders depending on connections and layout. Each block contains its execution order number. In (a) and (b) it is shown how the execution order is changed when the grey block is moved.



(a) (b) Sink tree of (a)

Figure 4: (a) is the graph representation of Figure 3d. Execution order numbers are within parentheses. In (b), the children are ordered in according to their positional order.

the set of blocks in a diagram, and compute the execution order based on this positional order. For the textual syntax, the positional order is simply the *declaration order* of the blocks. For the visual syntax, the positional order is defined as the *graphical order* based on the distance from each block to the origin in the following way:

Let $dist(v)$ denote the distance between a block v and the origin. We define the graphical order between two blocks A and B as follows: $\mathtt{A} \leq \mathtt{B}$ if and only if $dist(\mathtt{A}) < dist(\mathtt{B})$ or $(dist(\mathtt{A}) = dist(\mathtt{B})$ and $x_{\mathtt{A}} \leq x_{\mathtt{B}})$. To create an antisymmetric relation, we forbid two distinct blocks in a diagram to share coordinates, that is, the condition $\neg(x_A = x_B \wedge y_A = y_B)$ holds for all blocks A and B in a diagram where $\mathtt{A} \neq \mathtt{B}$. The visual editor makes sure this condition is satisfied.

Note that the graphical order and the declaration order must always be the same, in order to have a consistent semantics in the two syntaxes. Therefore, when a block is moved in the visual editor, the declaration order in the textual syntax may need to be changed, and vice versa. The coordinates can optionally be stored in the textual syntax, but are just secondary notation that does not influence the semantics. In case the coordinates are inconsistent with the semantics, the visual editor will discard them and use auto layout instead.

4 Execution Order

The connections in a diagram determine a partial execution order, and there are many ways of choosing a total order. In ABB's diagram language, a total order is chosen that follows a number of intuitive rules, as illustrated in Figure 3. For instance, blocks in a horizontal chain are executed in sequence, as illustrated in Figure 3a. Furthermore, it is the distance to the origin of the first block in a subchain that decides which subchain executes first, as illustrated by the grey block that has been moved in Figure 3b. If we have two unrelated chains as in Figure 3c, the upper chain is executed before the lower one.

The execution order becomes more complicated when a block has more than one predecessor. For example, in Figure 3d, block 6 has three predecessors. The execution number of block 6 will be determined by the predecessor with the highest execution number, in this case block 5. We say that block 6 is *sunk* by block 5, and we call block 5 the *sink pred* of block 6. The positional order is only used to determine the execution order between blocks that are sunk by the same predecessor, or that have no predecessor. This means that the position of blocks 6 and 4 is not

relevant for the execution order, since they are sunk by different predecessors.

We have formalized the computation of the total execution order based on a directed graph that models a diagram. In a diagram, block `A` can be connected to several ports on block `B`. These connections are represented by only one connection in the graph, since we are only interested in the dataflow between blocks. Figure 4a shows the graph representation of the diagram in Figure 3d, but with new block names.

The bold connections in Figure 4a represent which predecessor a block is sunk by. We can model these connections as a tree, which we call the *sink tree*, where the children are ordered by the positional order, as can be seen in Figure 4b. When we have this tree, we can easily compute the execution order by a depth first search (DFS), where children are visited from left to right. The resulting DFS number is the block's execution number.

4.1 Defining the Sink Tree

The blocks make up a directed acyclic graph (DAG). We extend this DAG with a root in order to handle blocks with and without predecessors uniformly. The new root r has no predecessor and has edges to all blocks with no predecessor in the original graph. Let $G_0 = (V,E)$ denote the original graph and let $pred(v)$ denote the immediate predecessors of v. The new graph G is then defined as follows.

$$G = (V \cup \{r\}, E \cup \{(r,v) \mid v \in V \wedge pred_{G_0}(v) = \emptyset\})$$

We define the sink tree by defining a block's parent in the tree, that is, the sink pred (sp) of the block. It is defined as follows:

$$sp(v) = \begin{cases} \bot & (pred_G(v) = \emptyset) \\ u \text{ where } u \in pred_G(v) \text{ and } \forall p \in (pred_G(v) \setminus \{u\}) : compare(p,u) \end{cases}$$

The first case is when block v has no predecessors in G, that is, when $v = r$. Hence, the root block r will be the root of the sink tree as well. The second case is when v has predecessors, that is, when $v \neq r$. In this case, the function returns the predecessor u with the highest execution number, that is, the predecessor with the highest DFS number. The predicate $compare(a,b)$ returns true if block a is executed before b. The sink pred u should execute after all other predecessors of v. The predicate $compare(a,b)$ is defined as follows, where we use the positional order ($po(v)$[1]) and the depth of a block in the sink tree ($d(v)$).

$$compare(a,b) = \begin{cases} po(a) < po(b) & \text{if } sp(a) = sp(b) \quad (1) \\ \text{true} & \text{if } a = sp(b) \quad (2) \\ \text{false} & \text{if } sp(a) = b \quad (3) \\ compare(sp(a), sp(b)) & \text{if } d(a) = d(b) \quad (4) \\ compare(a, sp(b)) & \text{if } d(a) < d(b) \quad (5) \\ compare(sp(a), b) & \text{if } d(a) > d(b) \quad (6) \end{cases}$$

[1] The function po maps a block to a natural number and preserves the positional order.

$$d(v) = \begin{cases} 0 & \text{if } pred_G(v) = \emptyset \\ d(sp(v)) + 1 & \text{else} \end{cases}$$

When comparing blocks a and b, we want to know if a will have a lower DFS number than b. One way is to find their lowest common ancestor, and compare its children that are ancestors to a and b. If the child that is ancestor to a is before the child that is ancestor to b, then a will execute before b. For example, in Figure 4b, if we want to compare the blocks `B` and `E`, then we want to compare the corresponding children to their lowest common ancestor (`A`), which are `B` and `D`. Since `B` is before `D`, then `B` will execute before `E`. We find the lowest common ancestor by walking up tree, using $sp(v)$, which is done in case 4-6. The cases 1-3 handle when the lowest common ancestor is found. In case 1, the positional order is compared. In cases 2-3, a is the parent of b or vice versa.

The definitions of *sp* and *compare* are mutually recursive. However, because *sp* only calls *compare* on predecessor blocks, and the DAG is acyclic, the computation will terminate.

4.2 Implementing the Sink Tree as Attributes

The sink tree described above has been implemented as RAGs in the JastAdd system, see Figure 5. Each function is implemented as a synthesized attribute. By writing `syn int Block.d()`, we define a synthesized (`syn`) attribute `d()` with return type `int` on nodes of type `Block`, where `Block` is an AST node type defined in the abstract grammar. For synthesized attributes, the equation defining the value can be written as a function body of ordinary Java code, and can use other attributes by calling their corresponding methods. In contrast to Java methods, attribute equations are forbidden to have side effects, and their values can be automatically cached.

Note that the attributes are a straightforward implementation of the mathematical definitions of *sp*, *compare* and *d*. The attribute `sp` is a reference attribute (an attribute denoting another AST node of type `Block`), `compare` is a parameterized attribute (comparing a Block with another Block), and `d` is a simple attribute of type `int`. Furthermore, in the definition of `sp`, the attribute `pred` is used which is the set of references to the predecessor blocks, according to the connections. The `pred` attribute is in turn defined using attributes defined in the name analysis, that use reference attributes to bind uses of names to their declarations. These definitions are not shown here, but similar RAGs can be found in, e.g., [Hed09].

5 Reusing Attributes in the Editor

The attribute implementation allows AST node properties to be reused during visual editing to provide useful semantic feedback. Since the attributes are defined declaratively and have no side effects, they can be reused without having to think about in what order they are computed. This means that we can access all attributes in the AST without bothering about what information (which attributes) they depend on. Thus, in contrast to a traditional compiler, there are no explicit phases of name analysis, type checking, etc. We will now give examples of how the attributes in PicoDiagram can be reused.

```
syn Block Block.sp() {
    Block u = null;
    for (Block p: pred())
        if (u == null || u.compare(p))
            u = p;
    return u;
}

syn boolean Block.compare(Block b) {
    if (sp() == b.sp()) return po() < b.po();
    if (this == b.sp()) return true;
    if (sp() == b) return false;
    if (d() == b.d()) return sp().compare(b.sp());
    if (d() < b.d()) return compare(b.sp());
    return sp().compare(b);
}

syn int Block.d() {
    return sp() == null ? 0 : sp().d() + 1;
}
```

Figure 5: Sink tree implemented as attributes

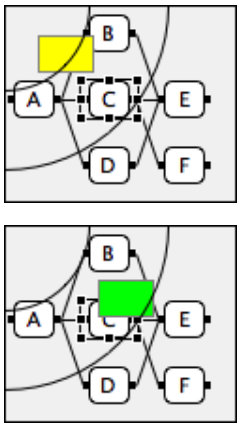
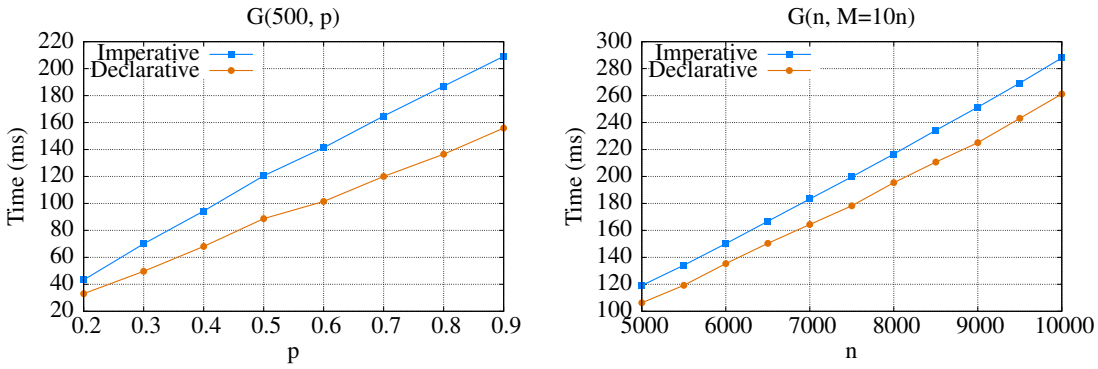


Figure 6: Feedback when a block is moved

Show Execution Order. One example is to reuse the attributes to show the execution order number for each block in the visual editor, as in Figure 3. If the user changes the diagram, then the execution order numbers are updated.

Move Block Feedback. Another example is to reuse the sink tree to provide feedback when the user moves a block. For instance, we have implemented visual feedback showing within what area a block can be moved, without changing the execution order. This can be seen in Figure 6, where two circle arcs are drawn to show the upper and lower bounds in terms of distances to the origin. The block is colored green as long as the block is moved within the bounds, and otherwise it turns yellow. To provide this feedback, the sink tree is used, where the siblings of the block in the sink tree define the boundaries. The sibling on the left will set the lower bound and the sibling on the right will set the upper bound. For instance, in Figure 4a, the boundaries of `C` are defined by `B` and `D`, as Figure 4b shows. Let $dist(v)$ denote the distance to the origin of a block v. If we want the execution order to remain the same, then the move of `C` must satisfy $dist(B) < dist(C) < dist(D)$. (Or if two blocks have the same $dist$ value, the x coordinate is compared instead, as was described in Section 3.2.)

Automatic Layout. For visual languages automatic layout is essential. For example, if the user opens a diagram without coordinates in the visual editor, the automatic layout should be constrained so that the execution order remains the same. We can use the sink tree to define such constraints, where each block is constrained by its siblings in the sink tree. If we have an ordered sequence of siblings $(s_1, s_2, ..., s_k)$, then the constraints between these siblings are $dist(s_1) < dist(s_2) < ... < dist(s_k)$. Another possibility is to use the sink tree itself as the layout of the blocks (or rotated 90° and mirrored to get the dataflow from left to right), since it conforms to these constraints.



(a) Generated with method $G(n,p)$ (b) Generated with method $G(n,M)$

Figure 7: Performance comparison between the declarative attribute implementation and an imperative iterative implementation of the execution order on randomly generated graphs.

6 Evaluation

We have evaluated the performance of the declarative attribute implementation of the execution order to show that it is feasible in practice. For comparison, we have also implemented an imperative algorithm that computes the execution order by iterating over the blocks. Both algorithms use the same underlying RAG-based declarative name analysis. Since JastAdd generates Java code, we need to be careful when measuring the performance due to non-determinism (e.g., garbage collection, JIT compiler, dynamic optimizations) in the virtual machine. We follow the methodology described by Georges et al. [GBE07], and we measure the steady-state performance, since editors are typically long-running applications. We only measure the computation of the execution order, excluding the time for name analysis and semantic error checking.

The comparison is made on randomly generated DAGs consisting of 500 to 10000 nodes, generated with the tool GGen [CMP+10]. Figure 7 shows results on graphs generated with the Erdős-Rényi methods $G(n,p)$ and $G(n,M)$, where n is the number of nodes. The first method generates a graph where each edge is present with a probability p, meaning that the total number of edges is around $pn(n-1)/2$. The second method generates a graph with M edges that are chosen uniformly from all possible edges. Confidence intervals have been omitted from the figures since they were barely visible. The results show that the performance is roughly linear, and that there is no problem in handling graphs of several thousands of blocks and with ten times as many edges as blocks. In practice, diagrams are usually much smaller, and likely with much fewer edges, so the method is clearly feasible in practice. It is also interesting to note that the declarative implementation is slightly faster than the imperative one, probably due to the on-demand attribute evaluation.

The benchmarks were performed on a computer running the operating system OS X version 10.8.2, with the processor 2.53 GHz Intel Core i5 and 8 GB memory. They were run on the Java HotSpot(TM) 64-Bit Server VM version 1.7.0_10 with 1 GB heap size.

7 Related Work

We focus in this paper on using reference attribute grammars to declaratively specify semantics, and to reuse these specifications to provide semantic feedback in visual editors. In contrast to some other specification-based systems, we do not (so far) generate the visual editor itself, but have hand-coded it with the use of the Graphical Editing Framework (GEF).

DEViL is a system that generates visual editors from high-level specifications and it is based on attribute grammars [SCK09]. DEViL uses an abstract structure similar to ours: it is based on object-oriented concepts like classes, subtyping and inheritance. A class can have attributes, and also references that can be used for building graphs, but where values have to be given when the object is created. Such attributes are called *intrinsic attributes* in JastAdd, and cannot be used for general attribute-based computations like JastAdd's reference attributes. For the visual specification, DEViL supports so-called visual patterns, which are predefined reusable implementations of common visual representations, such as lists, graphs, tables and line connections. These visual patterns are defined using attribute grammars and the language developer can adapt them by giving additional attribute specifications. DEViL also uses attribute grammars for code generation. It would be interesting to combine visual patterns with our system.

The Graphical Modeling Framework (GMF) in Eclipse, which has several similarities with DEViL, generates visual editors from specifications based on meta-models. GMF is built upon GEF and uses the Eclipse Modeling Framework (EMF) for the abstract structure. In addition to this, EMF models can be annotated with visual properties, which the tool EuGENia can generate GMF specifications from.

Bürger et al. [BKWA11] combined RAGs and metamodeling, and used RAGs to define the semantics for EMF models. They described the semantics for a simple imperative textual language and extended it with a visual state machine language. They used the tool EuGENia to generate a visual editor, and provided semantic feedback about the reachability for a state. It is, however, not clear if EuGENia could provide the feedback described in this paper.

While we focus on the semantics, others have focused on defining the syntax for a visual language with grammars. Examples include constraint multiset grammars [Mar94] and graph grammars [Min02, RS97]. In these examples, a parser can be generated from a grammar that can determine if a diagram is valid or not according to the syntax of the language, and even identify subdiagrams that are valid. One suggested benefit of parsing is that it allows free-hand editing, that is, the end-user is not restricted to always have valid diagrams - but can temporarily have invalid diagrams. Minas [Min02] combines parsers with graph transformations to generate editors that support both free-hand and syntax-directed editing. Furthermore, he has extended this work to generate editors based on meta-models, also supporting both editing modes [Min06]. Our approach contrasts to these approaches by using a simpler syntax that does not constrain the user very much, and instead define well-formedness criteria by specifying them using RAGs which can capture arbitrarily complex context-sensitive semantics. Violations of well-formedness criteria, for example connections that break type rules, can then be displayed as errors in the visual editor, much like compile-time errors can be displayed in textual editors. TIGER [EEHT05] is another system based on graph grammars, which generates syntax-directed visual editors. It uses graph transformation rules to define the syntax and possible (complex) editing operations.

Erwig [Erw98] proposed to use an abstract syntax for visual languages to define the semantics

on. Our work differs from his in that we use a tree as the base structure, instead of a graph, and we extend the tree to a graph using RAGs. We also define the semantics with RAGs. We think a tree is useful since it is straightforward to serialize and easy to add new textual syntaxes to. Moreover, the containment relationship in EMF can be used to form trees [BKWA11].

Modelica is a language for modeling and simulating physical systems, and for which there is a RAG-based compiler [ÅEH10]. It also has both a visual and textual syntax, but in contrast to PicoDiagram, its semantics does not depend on layout.

8 Conclusion

In this paper we have demonstrated how visual editors can provide semantic feedback to the user by reusing the semantics developed for a compiler. In our case study, we use a visual control language that has both a textual and a visual syntax. Both the batch compiler and the visual editor reuse the same semantic specification, and the textual syntax is used as the serialization format. The execution order is dependent on the visual layout, and we have demonstrated how it can be implemented declaratively using RAGs, based on an abstract syntax that does not rely on explicit coordinates. The mathematical formulation of this execution order, and its direct translation to RAGs, has allowed us to reuse these computations to provide semantic feedback to the user, for example, showing how a block can be moved without changing the execution order. We have also evaluated the RAG implementation, and compared it to an imperative implementation, demonstrating that the RAG approach to this problem is feasible in practice.

In the future, we would like to generate the visual editor from a high-level specification instead of hand-coding it, and we would like to investigate if earlier work can be applied to this problem. For instance, it would be interesting to combine RAGs with GMF for the PicoDiagram editor. In the current implementation, all semantic properties are recomputed for each edit, and we plan to add incremental evaluation [SH12] to be able to reuse unchanged values between edits. We would also like to add the possibility to edit a diagram both textually and visually at the same time, where changes are propagated between the views automatically.

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.

Bibliography

[ÅEH10] J. Åkesson, T. Ekman, G. Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming* 75(1-2):21–38, 2010.

[BKWA11] C. Bürger, S. Karol, C. Wende, U. Aßmann. Reference attribute grammars for metamodel semantics. In *Software Language Engineering (SLE 2010)*. Pp. 22–41. 2011.

[CMP+10] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, F. Wagner. Random graph generation for scheduling simulations. In *International ICST Conference on Simulation Tools and Techniques*. Pp. 60:1–60:10. 2010.

[DKV00] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices* 35(6):26–36, 2000.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. Pp. 134–143. 2005.

[EH07] T. Ekman, G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*. Pp. 1–18. ACM, 2007.

[Erw98] M. Erwig. Abstract Syntax and Semantics of Visual Languages. *J. Vis. Lang. Comput.* 9(5):461–483, 1998.

[GBE07] A. Georges, D. Buytaert, L. Eeckhout. Statistically rigorous java performance evaluation. In *OOPSLA 2007*. Pp. 57–76. ACM, New York, NY, USA, 2007.

[Hed00] G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*. 24(3), pp. 301–317. 2000.

[Hed09] G. Hedin. An Introductory Tutorial on JastAdd Attribute Grammars. In *GTTSE 2009*. LNCS 6491, pp. 166–200. Springer, 2009.

[HM03] G. Hedin, E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Sci. of Comp. Prog.* 47(1):37–58, 2003.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold. An overview of AspectJ. *ECOOP 2001*, pp. 327–354, 2001.

[Knu68] 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.

[KV10] L. C. L. Kats, E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA 2010*. Pp. 444–463. ACM, 2010.

[Mar94] K. Marriott. Constraint multiset grammars. In *Visual Languages, 1994. Proceedings., IEEE Symposium on*. Pp. 118–125. 1994.

[MHS05] M. Mernik, J. Heering, A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37(4):316–344, 2005.

[Min02] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[Min06] M. Minas. Generating meta-model-based freehand editors. In *Proc. of 3rd Intl. Workshop on Graph Based Tools*. Electronic Communications of the EASST, 2006.

[RS97] J. Rekers, A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing* 8(1):27–55, 1997.

[SCK09] C. Schmidt, B. Cramer, U. Kastens. Generating visual structure editors from high-level specifications. Technical report, 2009.

[SH12] E. Söderberg, 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.

[Wil04] D. S. Wile. Lessons learned from real DSL experiments. *Sci. Comput. Program.* 51(3):265–290, 2004.