

Program Visualization using Reference Attributed Grammars

Eva Magnusson and Görel Hedin
Department of Computer Science, Lund University
Box 118, SE-22100 Lund, Sweden
`{eva|gorel}@cs.lth.se`

Abstract. This paper describes how attribute grammars can be used to integrate program visualization in language-based environments and how program visualizations can be specified and generated from grammars. It is discussed how a general solution for a simple grammar can be reused in grammars for other specific languages. As an example we show how diagram generation for a very simple state transition language can be integrated in a more complex specific state transition language. We use an extended form of attribute grammars, RAGs, which permits attributes to be references to nodes in the syntax tree. An external graph drawing tool is used to visualize the diagrams. The solution is modularized to support reuse for different languages and exchange of the external drawing tool for different types of visualization.

1. Introduction

Program visualization is an important technique useful to gain understanding of the structure of a program. Meaningful visualizations can be built from several different types of elements (words, images, ..) but graph drawing is the most popular way to present structural relationships. One example is call-graphs where functions are presented as nodes in a directed graph and possible calls between functions as edges. Other examples are UML class diagrams, where design is expressed through graphical notations, and state transition diagrams used to visualize finite state machines.

In this paper we discuss integrating program visualization in language-based environments and how such program visualizations can be specified and generated from grammars. We deal with static code visualizations, i.e., visualizations of the program code, in contrast to, e.g., dynamic code visualization (visualizations of an executing program) and algorithm visualization. We have an interactive environment, APPLAB (APPLication language LABoratory) supporting language-based editing of the grammars of a language as well as language-based editing of programs in the language [1,2,3]. APPLAB is based on structure-oriented editing and reference attributed grammars [11] (an object-oriented extended form of attribute grammars).

The main representation of the program is an abstract syntax tree (AST), but a program visualization is often based on some kind of graph. We use reference attributed grammars to describe how these graphs can be

generated from the syntax tree. External visualization tools can then be integrated provided that they have an import mechanism for graphs from text files in some documented format. The generation of the text files on the format required by the tool is also specified using reference attributed grammars.

We show how it is possible to modularize the solution for a particular kind of visualization so that both the underlying programming language and the external visualization tool can easily be exchanged. In this article, we use state-transition visualizations as a running example and show how the solution can be reused for different state-based languages. A similar technique could be used for other visualizations, e.g. to obtain UML class diagrams for different object-oriented programming languages.

The rest of this article is organized as follows. Section 2 presents the environment architecture. Section 3 describes a general solution for a simple state transition language and section 4 gives an overview of how the representation for an external tool can be generated. In section 5 we show how the solution can be integrated in a more complex specific state transition language. Comparison of our approach to some related work is done in section 6. Section 7, finally, gives a concluding discussion of our technique and how it can be developed further.

2. Environment architecture

The environment architecture consists of two parts; a language environment supporting language specification and language-based editing of programs in the specified languages, and a visualization tool, i.e., an external graph drawing tool used for visualizing programs. In our experimental platform we use our interactive language development tool APPLAB as the language environment, and the tool daVinci [6] from University of Bremen as the visualization tool. See Fig. 1.

2.1 The interactive language development tool APPLAB

The language environment is implemented using our interactive tool APPLAB (APPLication language LABoratory) [1,3]. The main goal of APPLAB is to support interactive development of application-specific languages, allowing the user to simultaneously work on the language definitions and experiment with the resulting language. Changes in the language are immediately reflected in the program editor. The system is based on structure-oriented editing and an object-oriented extension to attribute grammars.

An attribute grammar [12] is an extension of a context-free grammar where a set of attributes $A(X)$ are associated with each non-terminal symbol X . A set of equations $E(p)$ are associated with every production p . There are two kinds of attributes, synthesized and inherited. Synthesized attributes are used to propagate information upwards in the syntax tree and inherited attributes to propagate information downwards. For each production p :

$X_0 ::= X_1 \dots X_n$, $E(p)$ should have equations defining all the synthesized attributes of X_0 and all the inherited attributes of X_i , $i= 1, 2, \dots n$.

In the extended attribute grammars used in APPLAB, the context-free syntax is modeled as an object-oriented inheritance hierarchy of node classes, where the leaf classes correspond to productions and their superclasses to nonterminals. The class hierarchy allows attribute grammars to be written in a compact way by using the inheritance hierarchy to avoid much of the repetition of attributes and equations that is otherwise common in classical attribute grammars [9].

The root of the class hierarchy is the node class ANYNODE which can be used to model behavior common to all nodes in the AST. Every node in the syntax tree is an instance of a subclass of ANYNODE. Equations defining attributes can be viewed as parameterless virtual functions. It is therefore possible to make a default definition of an attribute in a superclass and then override it in a subclass. APPLAB also supports the definition of virtual functions with parameters.

APPLAB makes use of object-oriented concepts to organize the specification, but in contrast to object-oriented programming languages the specification contains only declarative constructs (no assignments or other imperative constructs). The attribute evaluation method used is demand evaluation, a simple but general evaluation method based on recursion [14]: When an attribute value is demanded, the right-hand side of its defining equation is evaluated (similar to a function call) and this will in turn lead to the demand evaluation of the attributes used in that equation. Optimal evaluation is achieved by caching evaluated attributes in the AST and cyclic definitions of attributes can be detected at evaluation time by setting a flag for each cached attribute. In APPLAB, the user can demand an attribute value to be displayed or written to a file.

In addition to the object-oriented style of specifying the attribute grammar, APPLAB supports reference attributed grammars, i.e., the ability to let an attribute be a reference to an arbitrary node in the syntax tree [11]. Reference attributes are similar to ordinary reference variables in object-oriented programming languages in that they can refer to other objects and be used to obtain arbitrary linked data structures (including cyclic structures). However, reference attributes differ from reference variables by being defined declaratively by equations. This is in contrast to the usual programming language approach of writing an imperative mutating computation to obtain the linked structure.

Reference attributes are useful for describing arbitrary relations between nodes in a syntax tree, in addition to the syntactic (tree-structured) relations that ordinary attribute grammars support. For example, call graph relations, inheritance relations, and state-transition relations are easily described using reference attributes. A few built-in structured data types like dictionaries mapping strings to node references (NodeDictionary) and sets/bags of node references (NodeBag) have been added to APPLAB in order to allow such relations to be described effectively.

A language is specified in APPLAB in a document containing several grammar aspects, see also Fig. 1. The ABSTRACT aspect defines the abstract context-free syntax of the language and the CONCRETE aspect defines the concrete syntax (how to unparse an AST as text in a window). The OOSL (Object-Oriented Specification Language [1,3,10]) aspect defines an attribute grammar. An OOSL specification can be split in a number of modules thus textually separating attributes and equations for different purposes, e.g., static-semantic checking and code generation. Similar possibilities for modularizing the attribution specification is available also in non-object-oriented attribute grammar systems such as the Synthesizer Generator [19]. From an object-oriented viewpoint, the OOSL modules are orthogonal to the class hierarchy. Similar modularization techniques are available also for some object-oriented programming languages, in particular the fragment system for the BETA language [13] and in subject-oriented programming [8].

Fig. 2 shows an OOSL example of how to add an inherited attribute root referencing the root node in the AST to every node. The equation is an example of a so called collective equation which defines the value of an inherited attribute of all sons of a given type, in this case of any type [9]. The example also shows an example of an overriding equation: the equation in Program overrides the default definition in ANYNODE since Program is a subclass of ANYNODE.

2.2 The graph drawing tool daVinci

The external tool used for the graph visualization is daVinci [6], an interactive visualization system for drawings of directed graphs, developed at the Computer Science Department at the University of Bremen. An application program can access the operations of daVinci by using its API. The communication between daVinci and the application is realized with UNIX pipes. There is also a Graph Editor Application which is an interactive tool to create and modify graphs. The editor in this case acts as an application program which communicates with the daVinci API. Currently, we use the daVinci editor only to display graphs, not to edit them. Graphs can be loaded in the editor from text files with a special format, the term representation, described in more detail in Section 4.1.

After loading a graph into daVinci it can be processed in different ways. For example edge crossing minimization and edge bending minimization can be performed. It is also possible to create a survey view of the graph and zoom into different part of it and to change the orientation of the graph.

In our experimental platform, an attribute grammar is specified in APPLAB which defines the representation required by daVinci as a string attribute. Thus, to visualize a program, this attribute is evaluated and saved on a text file which is then loaded into daVinci, and displayed on the program window of the daVinci editor. See Fig 1. Our ambition is to improve the integration of the language environment and the visualization tool. Preferably

it should be possible to connect to the external graph drawing tool directly from APPLAB.

2.3 Obtaining a reusable visualization specification

In order to obtain a general reusable solution, it is useful to organize the specification of a visualization according to the following different aspects: First, the essence of the visualization can be specified by introducing node classes that match the main concepts in the visualization. For example, for a state diagram, the node classes could be State and Transition. Attributes of these node classes are introduced for modelling the essential properties of the visualization. We call this part of the specification the visualization front-end. Second, to tie this specification to a certain visualization tool, a visualization back-end module is introduced which specifies the computations needed to generate the representation required by the external visualization tool. If the visualization tool is exchanged, another back-end is written for that tool. Third, a visualization glue module is written which ties the front-end to the specific language to be visualized. Typically, the glue module can make use of a static-semantics module which defines the name analysis (identifier declaration/use sites) for the language at hand. To visualize another language with the same kind of diagrams, a new glue module is written. The front-end module can be reused for all languages and visualization tools. This module organization is shown in Fig. 3.

APPLAB currently supports this module organization, with the restriction that the front-end must use the node classes that correspond to states and transitions in the ABSTRACT syntax. As future work, we plan to generalize the module system of APPLAB in order to allow the front-end to introduce its own node classes, and let the glue module tie these node classes to the corresponding ones appearing in the ABSTRACT syntax.

3. Visualization for a simple state transition language

We will use state-transition diagrams as our running example. In this section, we introduce a very simple state transition language, TinyState, and the front-end and glue of our solution, i.e., how to specify the essence of the state-transition graph on which visualizations of programs written in TinyState are based, and how to tie this to the syntax of TinyState.

3.1 The language TinyState

The abstract grammar for a very simple state transition language, TinyState, is given in Fig. 4. Production (2) is a list production stating that a StateDecls consists of a number of StateDecl nodes. Production (5) is a construction production stating that TransitionDecl is a construction of three IDs (the name of the transition, the name of the source state and the name of the target state).

The concrete syntax of TinyState becomes evident from the example program of Fig. 5. The program can be visualized as a directed graph, where vertices correspond to states and edges to transitions.

3.2 Visualization front-end

The visualization front-end defines a representation of the state- transition graph that is independent of the programming language syntax and which is easy to traverse for the back-end. The representation is realized using reference attributes that link together declarations of transitions and declaration of their source and target states.

In every TransitionDecl node we add two attributes sourceState and targetState referencing the StateDecl nodes in the tree corresponding to the source and target states. Every StateDecl node has an attribute outgoingTrs defined to be a set of references to the transitions having the actual state as its source. There is also a corresponding attribute incomingTrs for transitions having the state as their target. In this way we get a description of the graph resembling an ordinary adjacency list representation which makes it easy to traverse. In Fig. 6 the connections between StateDecl nodes and TransitionDecl nodes in the AST for a small program are shown. The connections between nodes via reference attributes are illustrated with thick gray lines.

To define the attributes sourceState and targetState, detailed knowledge about the programming language syntax is needed, and the equations defining these attributes are therefore placed in the glue module. The attributes outgoingTrs and incomingTrs can then be computed in a syntax-independent way, using the values of sourceState and targetState. E.g., outgoingTrs can be defined by searching the AST for TransitionDecl nodes where sourceState references the actual state. A function outTrs is defined which recursively searches the tree for nodes matching this condition. To be able to start the search at the root of the AST, a reference attribute ASTRoot is introduced which must be defined by the glue module. The attribute incomingTrs is defined in a similar way. Fig. 7 shows the front-end module.

The outTrs function in Fig. 7 makes use of the foreach-construct in OOSL. In this case it is used to iterate over all the sons of an arbitrary AST-node. The same construct can be used to iterate over the elements in a NodeBag. (Despite its imperative appearance, the foreach construct is a declarative language construct equivalent to a special case of a tail-recursive function.) The function outTrs may seem unnecessarily complicated for our simple language. Since we know that all TransitionDecl nodes in the AST are sons of the TransitionDecls node iterating over these would be sufficient. Implementing this function (and others) in a more general way means, however, that we are able to reuse them when adding visualization aspects to other languages with completely different syntax tree structures.

We also declare two string attributes stateLabel and transitionLabel in the front-end module. They denote the text to be attached to nodes and

edges respectively in the visualization graph and are to be defined by the glue module.

3.3 Static-semantics of *TinyState*

It is often useful to base the glue module on the static-semantics module, because this module already contains the name analysis needed for the glue module. The static-semantic analysis of *TinyState* is very simple. One of its goals is to check that the names of states used in transition declarations are declared in the program. For this purpose an aggregate attribute `stateDict` (a `NodeDictionary`) is defined as a mapping from state names to references to their respective declaration nodes. Checking that a transition declaration is correct means checking that the names of the source and target states can be retrieved from the dictionary. The dictionary is an attribute defined in the root node of the syntax tree. All other nodes of the AST are given access to the dictionary via an attribute `root` referencing the root node and defined as was shown in Fig. 2. The definition of `stateDict` is shown in the table of Fig. 8.

The function `buildDict()` in node class `StateDecls` uses the `foreach` construct in OOSL. In this case a table is built containing association pairs (key,element) for all sons (all of which are `StateDecl` nodes) where key is the name of the son and element is a reference to the son node. If the same key is added more than once to a `NodeDictionary` the previous association is overridden. The table `stateDict` can therefore also be used when checking that all state names in a program are unique. To each `StateDecl` node an equation can be added where a lookup for the state name is performed. The corresponding element should refer to the actual state. If not, a violation of the uniqueness requirement has been detected. The complete static-semantics is available in a separate report [16].

3.4 The glue module

Fig. 9 shows the glue module. It should implement some of the attributes declared in the front-end according to Fig. 7. The attributes `sourceState` and `targetState` can be defined simply by doing a lookup in the dictionary `stateDict` defined in the static-semantics module. The `ASTroot` attribute can be defined simply using the `root` attribute (as defined in Fig 2). For `stateLabel` and `transitionLabel` there is little choice in *TinyState* but to define them as the names of the corresponding state and transition respectively.

4. Visualization back-end for the state transition visualization

Once the graph has been described in the front-end, by linking together `StateDecl` and `TransitionDecl` AST nodes via reference attributes, the representation required by the external graph drawing tool can be specified independently from the actual underlying language. In this section we give

an example of such a back-end, by showing how code is generated for output to the daVinci tool. The daVinci tool represents graphs as nodes and edges, and the goal of the back-end is thus to map the StateDecl-TransitionDecl graph of the program to the format for nodes and edges required by daVinci. This mapping is non-trivial because daVinci represents graphs as trees with special treatment of edges that cannot be mapped to a tree and special treatment of cyclic structures in the graph.

4.1 The daVinci term representation

When graphs are loaded in daVinci a special format called the term representation is used. The term representation is defined by a context-free grammar. A term is a structure of type `parent[child1, child2,...]`. Brackets are used around a list of elements of the same type. The scheme is applied recursively. The term representation is plain text, so it can be created manually using a text editor. Typically, however, it is created automatically by some application program. In our case the input to daVinci is created by defining a string attribute of the program to be visualized.

Identifiers and *references* are used to identify daVinci nodes and edges. If a child node has more than one parent the subgraph of the child appears only once in the term representation (as a child of one of the parents). This subterm is given an identifier, the identifier of the child node. The other parents have only a reference to this identifier. For example the node C in the graph of Fig. 10 has two parents A and B. When generating the code the node A will be treated as parent of C and when visiting C on a traversal coming from A we will continue recursively to generate the code of the subgraph of C. Coming from B, however, we will stop the traversal at C and just return the code of a reference to the child C.

One task of the back-end is to generate unique identifiers for all daVinci nodes and edges. This is done by adding an integer attribute `prefixNbr` to all nodes in the AST and defining this attribute as the number of the node when traversing the tree in prefix order starting with number 1 in the root node. The relationship between the number of a node and the number of its parent can be expressed as

$$\text{prefixNbr} := \text{parent.prefixNbr} + 1 + \text{nbrOfNodesInLeftSiblings}$$

where the last term denotes the total number of nodes in the subtrees rooted in the left siblings of the actual node. This number can be computed using auxiliary functions implemented in ANYNODE, in a manner completely independent of the underlying programming language. Since daVinci requests its unique identifiers to be strings, another string attribute `nodeId` is defined in each node. Its value is simply the `prefixNbr` attribute translated into its corresponding string. See the appendix for the specification of these computations.

The term representation uses attributes to specify the visualization of individual daVinci nodes and edges. We call these attributes daVinci attributes

to distinguish them from attributes of an attribute grammar. All daVinci attributes have default values. The following example shows daVinci attributes for a graph node which should be drawn as a box (a default shape value) with blue background and text "hello" written using the default font ("a" is the constructor for string pairs defining daVinci attributes in the term representation):

```
[a("OBJECT","hello"), a("COLOR","blue")]
```

All daVinci attributes that need to be defined have their corresponding attributes in our grammar. We have chosen to draw the nodes as ovals and edges as lines with an arrow pointing to the target state.

4.2 Code generation by graph traversal

The daVinci term representation (in the following called simply the code) can be generated by using information propagated along the reference attributes describing the graph, corresponding to depth-first graph traversal. There are however two problems which need to be dealt with.

The first problem concerns cycles in the graph. In an imperative language you usually perform depth-first traversal by adding an extra boolean attribute "visited", initially false and changed to true when visiting the node for the first time. In a declarative language it is not possible to change the value of an attribute once defined. The usual solution is to use sets to keep track of which nodes to visit next and which nodes to avoid to visit again. A simpler technique for the problem at hand is to avoid cycles by inverting edges and then draw them reinverted. Since all nodes in the AST are numbered (the prefixNbr attribute) we can introduce an order between states. We define a state S1 to be declared before another state S2 if $S1.prefixNbr < S2.prefixNbr$. An attribute inverse is added to TransitionDecl nodes. If the source of a transition is not declared before its target the value of inverse is true, otherwise false. In the code generation phase a transition where inverse is true will be treated as a transition from its target state to its source state. One of the daVinci attributes for edges specifies the way the edge should be drawn. If inverse is true then the corresponding daVinci attribute is defined to draw the edge inverted i.e. it appears with its original orientation in the visualization. Selfedges need special treatment.

When cycles are removed the code can be generated by appending the code representation of all StateDecl nodes with indegree 0. The code of a StateDecl is constructed by appending the code of all transitions in its outgoingTrs attribute where inverse is false and all transitions in its incomingTrs attribute where inverse is true. The code of a transition is in turn in principle the code of its target state. daVinci attributes are inserted in appropriate places as stated by the grammar of the term representation. Thus the generation of the code of a state corresponds to a depth-first traversal of the graph starting in the actual state.

The second problem originates from the requirement to describe a sub-graph only once in the term representation as explained in section 4.1. For this purpose we add an attribute `theTransition` to each `StateDecl` node defined to be a reference to one of the transitions having the actual state as its target. In the code generation we continue to recursively visit and generate the code of a target state node if the transition node being treated equals its `theTransition` attribute. Otherwise we just return the reference code (the unique identifier) of the target state node.

An outline of the specification is shown in the appendix. The complete back-end specification is available in [16].

5. Reusing the visualization specification for a more complex state transition language

The state-transition visualization specified by the front-end and back-end modules can be used for any state-transition language simply by exchanging the glue module. In another research project at our department, an application-specific language has been developed to support executable state-transition based specifications for devices communicating over short-distance radio. This project is done in cooperation with Ericsson Mobile Communications [7]. We have used this language, `ExSpecState`, as an example of how to integrate the diagram generation in a specific state transition language. In this section we will discuss the glue module for `ExSpecState`.

5.1 Differences between the languages

The ABSTRACT grammar of `ExSpecState` contains approximately 90 productions. It comes equipped with OOSL grammar aspects for name and type analysis. As a state transition language it differs from `TinyState` in two ways. The states are hierarchical and the transitions have no names. An excerpt from the ABSTRACT grammar showing only the productions affecting state and transition declarations is given in Fig. 11. and part of a program using `ExSpecState` is shown in Fig. 12.

5.2 Integrating the diagram generation

The reuse of the front-end and back-end modules currently relies on that all programming languages use the names `StateDecl` and `TransitionDecl` for the non-terminals modelling states and transitions.

In the glue module for `ExSpecState`, definitions for the attributes `sourceState` and `targetState` in a `TransitionDecl` node are added. `targetState` could be defined using the lookup facilities provided by the name analysis of `ExSpecState` (as was done in the glue module of `TinyState`). In fact it doesn't need to be looked up since each `Use` node is already equipped with an attribute (`decl`) referencing its corresponding declaration. The `sourceState` is in `ExSpecState` a reference to the declaration of the state in which

the transition is declared i.e. we have to look for the closest ancestor node of type `StateDecl` in the AST. The principal part of the glue module for `ExSpecState` is shown in Fig. 13 with equations for `sourceState`, `targetState`, `stateLabel`, `transitionLabel` and `ASTroot`. The attribute `parentState` used in the definitions is declared in `StateDecl` nodes and defined to reference the closest ancestor of type `StateDecl` in the AST. If there is no such ancestor it references the ancestor of type `Process` instead. Labels for states are in principle defined by appending the labels of its parent states (the symbol "&" is used for appending). If a process named P contains a declaration of a state named S1 which in turn contains a declaration of a state named S2 then the label of S2 will be P_S1_S2. For transitions the label is the comment attached to its declaration if present otherwise the name of the event causing the transition.

6. Related work

Our visualization technique can be characterized as follows:

- *static code visualization*: the scope of our technique is restricted to static code visualizations, i.e., visualizations that can be derived from the program code (in contrast to dynamic visualizations like execution visualization and algorithm animation).
- *open system*: visualizations are not built into the environment, but can be added as desired.
- *declarative specifications*: visualizations are specified using a declarative formalism, rather than explicitly programmed.
- *language independent*: the visualizations can be specified independently of the programming language used, and reused for different programming languages by specifying different glue modules
- *visualization tool independent*: different visualization tools can be used by specifying different back-end modules

There are many systems that have support for some kind of program visualization. However, most systems are not open, but provide support only for a set of built-in visualizations. They are usually also language dependent and provide support only for a predefined set of programming languages. One example is Panorama, a visual environment for Java/C/C++ which supports visualizations like call graphs and flow diagrams [17]. Other examples include Rational Rose [18] and TogetherJ [24]. These latter systems provide support not only for program visualization, but also for visual programming, i.e., the possibility to edit the diagrams. They support round-trip engineering where the user can edit diagrams with auto-updating source code and also edit source code with auto-updating diagrams. Dedicated visual programming environments include Prograph [4]. Language-based approaches to visual programming includes graph-grammar based environments, such as Progres [22]. A fundamental difference between these tools and ours is

that they use graphs as the main program representation, whereas in our approach the main representation is an abstract syntax tree described by a context-free grammar.

Pavane [20] is a tool that, like ours, takes a declarative and language-independent approach to visualization. However, the scope of Pavane is algorithm animation rather than static code visualization. The animator defines a mapping from program states to graphical objects. For some languages, like C++, some annotation of the program code is needed.

In [21], another language-based approach for using attribute grammars for visualization is described. However, this approach is restricted to tree-structured visualizations of syntax trees. The system integrates the language-based editor generator CENTAUR with a visualization tool FIGUE which is capable of displaying trees specified as Lisp lists. An example of its use is in visualizing mathematical formulas in their standard mathematical form. Attribute grammars are used, but only in the integration process of the tools. A given visualization can be reused for all languages specified in CENTAUR but the only structural aspect of a program that can be visualized is its abstract syntax tree. The solution seems to closely couple the tools with no intention of possible exchange of the visualization tool.

A language independent program visualization technique is described in [5]. Control structure diagrams (CSD) are generated automatically from source code. CSD diagrams add some graphical notations to pretty-printed source code in order to depict control structures and levels of nesting. The tool works in two phases. During the first phase markup tags are inserted in the source code to identify all control structures. In the second phase the tags are used to render the visualization. The renderer is completely language independent but a new tagger must be developed for every language. The separation of a language dependent phase from a language independent one resembles our modularization technique. The scope of the visualization is restricted to CSD diagrams, and diagrams with arbitrary relationships between program structures can thus not be handled.

7. Conclusions and future work

We have described a technique to integrate visualizations in language-based environments and how they can be specified declaratively in RAGs. We have also shown how the solution can be modularized to facilitate reuse for different programming languages and exchange of the external drawing tool.

Attribute grammars allow specification of context-sensitive aspects of a language such as semantic checking and code generation. The specifications are declarative and thus potentially clearer and more concise than imperative code since they only state facts about the final computation results and not the order of computation. The extension of canonical attribute grammars to RAGs makes it easy to define grammar aspects where non-local dependencies play an important role. An example is the visualization aspects as shown

in sections 3 and 4. Reference attributes permit a clear and concise way of describing the non-local dependencies in the AST that constitute the graph on which the visualization is based. Information can be propagated along the reference attributes describing the graph structure thus facilitating the generation of a correct representation for an external drawing tool.

For the definition of an individual attribute, one can always argue if an imperative procedure or a declarative function is easiest to understand. This was touched upon in section 4.2 where different techniques for handling cycles in the graphs were discussed. In principle, it would be possible to allow imperative definition of an attribute, provided this code does not produce any net side effects (i.e., side effects that remain after execution of the code). To support such imperative specification in a safe way could be a topic of future work.

It is straightforward to express a solution in general terms in APPLAB. Rather than using information about the structure of the syntax tree for a certain language, a more general approach can be taken by representing the important structures using reference attributes. This allows the front-end and back-end of the specification to be reused for different programming languages.

The APPLAB specification language currently supports modularization by allowing attribute definitions for a certain aspect of the grammar to be textually separated from other grammar aspects of the language being specified. As mentioned in section 2.3, a generalization of the module system is needed to make the front-end of our solution completely reusable for all programming languages. Currently, the essence of the front-end is reusable but relies on grammars to use the same names for its node classes. We plan to generalize this by extending the OOSL formalism with a possibility to declare syntactic part objects, similar to part objects in BETA [15] or anonymous inner classes in Java [23]. The back-end is concerned solely with the generation of a proper representation for an external tool. Using different tools for different kinds of visualizations can thereby be achieved by exchanging this module.

In the near future, we also plan to improve our tool integration so that visualizations in external drawing tools can be opened and updated more conveniently; in the current solution the user has to print an attribute to an explicit text file and start the drawing tool by a shell command.

A more long-term challenge is to try to extend the technique so that external visualization tools that support editing, like daVinci, can be used for actually editing the visualized program, and propagate those edits back to the original syntax tree. Preferably, the external tool should include an event propagation mechanism so that each individual editing step could be propagated back to APPLAB. A main challenge in making such integration work is to devise a mechanism that allows the change of a reference attribute value to induce a corresponding change to the AST. For example, changing an edge in the visualization graph means changing the value of reference attributes in the AST. The proper change of the AST to make it consistent

must then be found.

Acknowledgments

We are grateful to the anonymous reviewers for constructive comments and to Mathias Haage for providing us with pointers to various work on visualization. This work was partly supported by NUTEK, the Swedish National Board for Industrial and Technical Development.