Priority Controlled Incremental Attribute Evaluation in Attributed Graph Grammars*

Simon M. Kaplan and Steven K. Goering
Department of Computer Science
University of Illinois

Abstract: We show how incremental attribute change propagation can be efficiently done on Graph Grammars. The order of evaluation is controlled by priority numbers assigned to each attribute vertex. The priority numbers are assigned in linear time as directed by static grammar analysis. We also develop a powerful set-theoretic based embedding description format, and introduce the idea of `roles` as a mechanism to aid in graph grammar engineering. Roles are also a powerful aid to static analysis of grammar properties. A key to our approach is the uniformity gained by collapsing the substrate (structure) graph and the attribute dependency graph into a single directed graph structure. Related groups of attributes are simultaneously replaced by *cluster rewrites.*

1 Introduction

State-of-the-art program development environments use a structured representation for programs rather than a text stream. Such a representation allows the tool to provide incremental feedback to the user concerning the syntactic correctness of his code, and also to check that semantic constraints on the program (for example type-correctness) are met. Usually this representation is a tree [15] [2] [13]; a drawback of such a representation is that the tools do not seem to scale up to supporting more than programming-in-the-small. To correct this deficiency, graphs have been proposed as an alternative structure [9] [12] [1].

We are developing a family of tools which use *attributed graph grammars* to define the legal structures that can be built using the tools and to specify the constraints on the structures. We refer to "structures" rather than "programs", because our graph-grammar based tools support phases of the software development cycle other than programming-in-the-small [8]. An example we will use throughout this paper is that of *module interconnection structures.* A graph grammar is analogous to a string grammar, except that the grammar generates graphs rather than parse trees. Attribution is a well-know mechanism for describing semantic constraints on sentences in the language of a grammar [11], but research on the efficient evaluation of attributes has largely been confined to the case of string grammars [16] [10].

The purpose of this paper is to define an attributed graph grammar model and show how to perform efficient incremental attribute evaluation on the graphs constructed from a grammar that meets certain simple restrictions. Testing if a grammar meets these restrictions can be performed in polynomial time.

We are not the first to suggest the use of graph grammars in software development environments; The IPSEN project [12] [3] has investigated generating tools from graph grammars; in this project semantic constraints are specified using action routines [13]. A major problem with action routines is that they are

*Supported in part by the National Science Foundation under grant CCR-8809479 and by the AT&T Illinois Software Engineering Project. For more information contact Simon M. Kaplan, Department of Computer Science, University of Illinois, Urbana, Illinois 61801, USA. *email:* `kaplan@a.cs.uiuc.edu` *or* `uunet!uiucdcs!kaplan`.

not declarative, making them much harder to write and forcing the programmer to define explicitly actions to be taken in the face of editor actions such as insertions, deletions, etc. A declarative notation such as an attribute grammar does not have such problems [14].

Gottler [4] [5] has investigated the use of attribution of graph grammars to help in layout for diagram editing, but does not address the question of attribute evaluation.

Others have investigated environments in which the internal representations are attributed graphs, but do not use grammars to impose structure on the graphs [1]. Their work introduces the idea of using *priority numbering* to schedule attribute evaluations, but because of the lack of structure in the graphs, checking of semantic constraints after an edit cannot even be guaranteed to complete in polynomial time. Further, because of the lack of structure in their graphs, it is not clear how useful their approach will be; there is no way to describe abstractions on structures, or generate tools automatically from formal specifications.

While our approach is greatly influenced by this work, especially the idea of priority numbering, our use of grammars to impose structure on the graph allows a great improvement on these results; the cost of renumbering the graph after an edit is restricted to the size of the subgraph inserted during the rewrite, and the worst-case cost of the attribute evaluation is $O(|\ \mathit{Influenced}\ | \cdot \log(|\ \mathit{Influenced}\ |))$, where *Influenced* is the set of attributes which receive a new value as a result of the attribute evaluations, or are the immediate descendents of such attributes.

This paper makes several contributions to the theory and practice of building environments for software development. It proposes a powerful model of attributed graph grammars, and a new methodology of defining embeddings in rewrites. It introduces the idea of *roles* as a mechanism which aids in the engineering of the graph grammar and permits efficient attribute evaluation. It gives algorithms for the efficient evaluation of constraints on the structures provided the grammar meets certain properties, and describes tests to determine if the grammar does indeed have these properties.

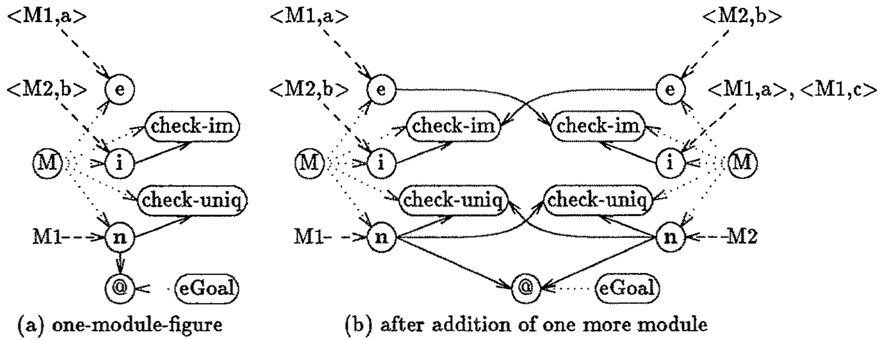
The remainder of this paper is structured as follows. We begin in section 2 with an informal example of the use of attributed graph grammars to describe the module interconnection structure in a large program. Section 3 introduces our graph grammar model. Section 4 describes how to incrementally reevaluate attributes after an edit. Details of the static grammar analysis algorithms are deferred to an appendix.

2 An Introductory Example

We illustrate our ideas by considering a *module interconnection structure*. Each module has a name, *imports* sets of variable names from other modules, and in turn *exports* a set of variable names. For simplicity we ignore the bodies of modules, and assume that both imports and exports are sets of tuples (the name of the module and the variable). It is trivial to rewrite the syntax of "real" languages to this form. The example will check two semantic constraints: all module names must be unique, and any variable imported from a module must be explicitly exported by that module.

We want to build a structure that will represent the interconnections among modules, and modify it each time a module's interface is changed or a new module is added to the program. After modifying the structure, we will need to check that the semantic constraints are still enforced. Figure 1 shows two sample interconnection structures; part (a) shows a single-module structure, and part (b) the structure after addition of a module.

We represent a module by a *cluster* of six vertices. One vertex is labeled by the symbol `M` and acts as a *center* for graph rewriting. Each of the other five vertices hold attribution information, including the name of the attribute, the *attribution rule* (i.e. function used to compute its value), the last value computed,



(a) one-module-figure (b) after addition of one more module

Figure 1: Example Graphs

and some housekeeping information. Three of the attributes represent the module name (`n`), its import list (`i`) and its export list (`e`). In each case the attribution rule to compute the attribute value is the constant function returning the appropriate information, shown in the diagrams as a literal connected to the appropriate vertex by a dashed line. The final two attribute vertices (labeled `check-im` and `check-uniq`) represent the computation necessary to check the semantic constraints. The operations are not shown to save space; they simply verify the conditions stated above. The center of the cluster is connected to all the attribute vertices by *cluster edges*, shown as dotted lines.

Edges are placed to the `check-im` attribute of each module to reflect the import/export dependencies among modules and other edges are placed to connect all the name attributes to the `check-uniq` attribute of each module in order to check that module names are unique.

Figure 1(a) shows a simple structure with one module cluster. The module has name `M1`, imports `b` from `M2` and export `a`. The `n` attribute vertex is connected to the `@` attribute vertex of a `eGoal` cluster. This is used in expanding the graph and will be described later. Note that the `check-im` attribute vertex is connected to no other modules; thus, the importation of `b` from module `M2` will be in error. Because there are no other modules, the uniqueness constraint is satisfied.

Figure 1(b) shows the graph after addition of a second module, `M2`, which exports `b` and imports `a` and `c` from `M1`. Edges are placed from `M2`'s `n` vertex to the `check-uniq` vertex of every other module cluster, and from every other module's `n` vertex to `M2`'s `check-uniq` vertex. Edges are also placed from the `e` vertex of any module exporting variables which are imported by `M2` to `M2`'s `check-im` vertex, and from the `e` vertex of `M2` to the `check-im` vertex in the cluster of any other module exporting from it.

In this instance, the constraint on importation is satisfied for `M1`, but is not satisfied for `M2`, as `M1` does not export `c`. Further modifications to the structure would be needed to satisfy this constraint.

The rewrite of the graph just described is controlled by a `graph grammar`. The grammar for the module interconnection structure is given in figure 2. Each production consists of three parts: A name (with parameters), a goal and a bodygraph. The parameters are constant functions and are bound to the attribution rules of some of the attribute vertices; in our example, the module name, import and export lists are given values in this way. The goal is a symbol, and the bodygraph is analogous to the body of a production in string grammars. Each attribute vertex in the bodygraph is labeled by several things, including an attribution rule (either a constant or some other function), a name, and a *role*, R_i. Rewriting replaces a cluster by a bodygraph, as follows:

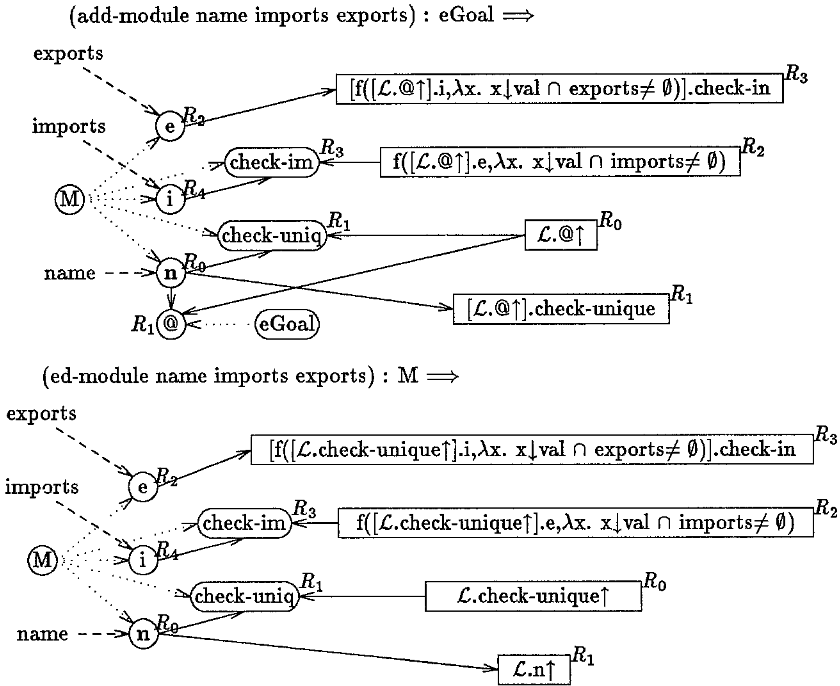


Figure 2: Module Interconnection Grammar

- The symbol labeling the center of the cluster is identified. This must the be same as the goal symbol of the production to be used.
- The vertices in the cluster are identified by following the cluster (dotted) edges.
- The *neighborhood* (all clusters adjacent to the cluster about to be rewritten) is identified. Two clusters are adjacent if there is an edge from some vertex in one to some vertex in the other.
- The cluster is removed from the graph, along with all incident edges.
- The bodygraph of the production is instantiated into the graph.
- The vertices in the instantiation of the bodygraph are *embedded* into the graph by placing edges from (to) vertices in the bodygraph to (from) vertices in the graph. The bodygraph can only be connected to vertices in the neighborhood identified above.

Embedding is controlled by *embedding expressions*, shown in rectangular boxes in the figure. An embedding expression, when evaluated, returns a set of vertices in the neighborhood to which edges may be placed. Embedding expression and attribute vertices are labeled by *roles*. Roles are used to engineer the grammar, and play an important part in static grammar analysis. To see how the embedding expressions are used, consider the expression "$f([\mathcal{L}.@\uparrow].e, \lambda x.x\downarrow val \cap imports \neq \emptyset)$". It evaluates to the set of attribute vertices which export variables that are imported by the new module being added. This is explained in detail below.

Figure 1(b) is derived from figure 1(a) by applying the production `add-module`. The `eGoal` cluster is rewritten to a new module (`M2`), along with a new `eGoal` cluster, which can later be further rewritten to add more modules. Editing a module (for example, to add `c` to the export list of `M1` would involve rewriting

the cluster for `M1` by the `ed-module` production. Because the placement of edges in the graph can change if a module is edited, such edits have to be viewed as rewrites on the structure.

Thus far we have explained how a module can be represented in a graphical structure, and how graph rewritings can be used to connect each module to just the set of modules from which it imports or to which it exports. We now describe how constraint checking is actually performed. It is done by evaluating the attributes of each cluster. Evaluation terminates when the value stored at each attribute vertex is equal to the result of the execution of the attribution rule for that vertex. A poor choice of evaluation order could lead to an attribute being reevaluated many times; the challenge of incremental attribute evaluation is to choose an evaluation ordering that restricts an attribute to being evaluated only when the evaluation will yield the final value.

To achieve this, we assign to each attribute vertex a *priority number*, and perform analysis on the grammar which determines the priority order for the attributes in each production (roles are used to help determine this order). When we rewrite the graph, we interpolate new priority numbers for each vertex added to the graph between the numbers of the old vertices in the graph. The priority numbers capture all transitive dependencies among the attributes. Attributes are then evaluated in priority number order. A priority queue is maintained, initially holding only the newly created attributes. Each time an attribute is evaluated, if the result of the attribution rule differs from the previous value, then the descendants of the attribute vertex are added to the queue. In this way, each attribute is evaluated only when all the attributes on which it depends have their correct final value, and only attributes *influenced* by the edit are evaluated. (An attribute is said to be influenced if its attribution rule returns a value different from that of the prior evaluation, or if an immediate ancestor in the graph had a different value).

3 Attributed Graph Grammar Definition

In the notation below, we use x_Y to represent the field x of tuple Y. Where the tuple is obvious from context the subscript is omitted to reduce clutter.

Definition 1 *A* World *is a tuple* $W = < sym, attr, role, val, fns, A >$ *where:*

- *sym is a finite set of symbols;*
- *attr is a finite set of attribute names;*
- *role is a finite set of roles;*
- *val is an arbitrary domain;*
- *fns is a finite set of functions. Each element of fns takes zero or more values from val and produces one or more values of val, i.e. is a multivalued function of multiple arguments; and*
- $A : sym \rightarrow \wp(attr)$ *is a function that associates a set of attribute names with each symbol, where* $\wp$ *is the powerset operator.*

Sym, *attr*, and *role* are name sets that will be used to label grammar productions and graph vertices. To simplify description we assume these to be pairwise disjoint. *Val* is the domain of legal values that attributes may have. We do not preclude the use of attributes of different types, *val* is the sum domain of all relevant types. *Fns* is the set of attribution rules to do the computations. Finally, A declares attribute names for each symbol.

$< e4 > ::=$	$\mathcal{L}$	base set
	$\mid < e4 > . < \text{attrname} >$	narrow
	$\mid < e4 > \uparrow$	follow
	$\mid$ '[' $< e4 >$ ']'	widen
	$\mid \texttt{f}(< e4 >, < \text{pred} >)$	filter
	$\mid < e4 > \cup < e4 >$	union
	$\mid < e4 > \cap < e4 >$	intersection
	$\mid < e4 > - < e4 >$	set difference

<pred> is a predicate of one vertex parameter

operator	type	description
$\mathcal{L}$	$1 \rightarrow$ set:symb	singleton set containing symbol vertex of the goal cluster
.	set:symb $\times$ attrname $\rightarrow$ set:attr	follow C-edges to attr vertices of given name
$\uparrow$	set:symb $\rightarrow$ set:attr set:attr $\rightarrow$ set:attr	follow all D-edges incident on any element of the set (either to source or target)
[ ]	set:attr $\rightarrow$ set:symb	return all clusters to which one or more attrs being considered belongs
f	set:a $\times$ predicate $\rightarrow$ set:a	return all elements that satisfy predicate
$\cup, \cap, -$	set:a $\times$ set:a $\rightarrow$ set:a	standard set operations

Notes:

- 'set:symb' means "set of symbol vertices", 'set:attr' means "set of attribute vertices", and 'set:a' means "set of vertices of one of the two kinds".
- Predicates for filters take a single vertex as a parameter. They are allowed reference the vertex label (and thus symbol name, attribute name, role, value, etc.) only. Syntax to reference a field is '$\downarrow < \textit{fieldname} >$'.

Figure 3: Syntax and semantics for Edge-End Embedding Expressions

of the production. The rewrite finishes by embedding the daughter graph into the *host graph* (the original with the cluster removed). Embedding is restricted to the distance one neighborhood of the goal cluster (1-NCE) [6]. Daughter graph vertices may only be connected to other daughter graph vertices and to vertices that were originally in the neighborhood of the goal cluster (i.e. adjacent to the goal cluster). We provide a language, called *Edge-End Embedding Expressions* (E^4), to refer to particular subsets of the vertices in the neighborhood of the goal cluster.

Definition 3 Edge-End Embedding Expressions *have syntax and semantics as shown in figure 3.*

As examples we explain some of the expressions used in the rectangles of the Module Interconnection grammar of the previous section.

The expression $\mathcal{L}.n \uparrow$ is evaluated by taking the the n attribute of the goal cluster, i.e. $\mathcal{L}.n$, and following all D-edges incident on it out of the cluster. This expression is used to embed the n attribute of the replacement cluster (for the module editing production). We can determine by inspection that all D-edges are out-directed from it, and that is exactly how the embedding expression is being used to replace D-edges, so this will do an "identity embedding", that is the replacement cluster will be connected identically as the old cluster was connected at this attribute vertex.

The rolename acts as an additional filter to control embedding. In the example grammar, rolenames nearly duplicate attribute names (although several different attributes, `c` and `check-uniq`, take role R_1 indicating that they accept 0 or more module names). In the example productions the rolenames provide

3.1 Attributed Graphs

Definition 2 *A* graph *with respect to world W is a tuple $G = < Sv, Av, sl, al, E >$ where:*

- *Sv is a finite set of symbol vertices;*
- *Av is a finite set of attribute vertices;*
- *$sl : Sv \rightarrow sym$ is a labeling function assigning a symbol to every symbol vertex;*
- *$al : Av \rightarrow attr_W \times role_W \times Q \times fns_W \times val_W$ is a labeling function assigning a 5-tuple to every attribute vertex. For every vertex $v \in Av$, the label $al(v) = < a, r, p, o, v >$ gives its name, role, priority (a rational number[1]), attribution rule, and value respectively;*
- *E is a finite set of (directed) edges, where each edge is specified as a source vertex and a target vertex. An edge is called a* dependency *(D-edge) if its source is an attribute vertex and a* cluster edge *(C-edge) if its source is a symbol vertex.*

G is well-formed *if:*

- *The target of every edge is an attribute vertex;*
- *For each symbol vertex v with label $sl(v)$, let B be the set of all attribute vertices adjacent to v through C-edges. Then no two elements of B have the same name and the set of all the names of elements of B equals $A_W(sl(v))$;*
- *Every attribute vertex is the target of either zero or one C-edge;*

For convenience the term graph *will imply well-formedness below unless otherwise stated.*

Informally, a graph is a set of *clusters* (attribute vertices tied to a central symbol vertex with out-directed C-edges, i.e. dotted arrows) and some attribute vertices that are not part of any cluster; plus D-edges between the attribute vertices (drawn by continuous arrows). Example clusters are centered around the symbols `M` (with attributes `e`, `i`, `n`, `check-in`, and `check-uniq`) and `eGoal` (with attribute `@`).

If a symbol s has a set of attribute names $A(s)$ associated to it, by the second requirement every vertex instance of s has attribute vertices of those same names associated with it *via* C-edges. The vertex s and its attribute vertices (and the C-edges connecting them) are a cluster. By the third requirement, clusters are disjoint, but do not necessarily cover the graph.

We allow attribute vertices to be part of no cluster as a convenient shorthand for building an extra (dummy) symbol vertex and making them all a cluster around this. The purpose that symbol vertices serve is as rewrite points, so any vertex analogous to a terminal symbol can be omitted to simplify the grammar. Any attribute vertices that need to remain are then not part of a cluster.

We define adjacency with respect to clusters as the set of all clusters (and attributes not part of any cluster) where some vertex in them is adjacent to some vertex in the original cluster.

As part of each attribute vertex label there is a priority, an operator, and a value. These are concerned with attribute evaluation so we defer full discussion of them until section 4. The rest of this section describes how graphs are rewritten by grammar productions.

3.2 Graph Rewriting

Productions are used to control *cluster rewriting*. Starting from a symbol vertex, all C-edges are followed to identify (in linear time) the cluster. This is replaced by the *daughter graph* instantiated from the *bodygraph*

[1] We make the assumption that operations on rational numbers can always be done in unit time. [1] justifies this assumption and discusses its implications in depth.

visual clue as to meaning (or would if space permitted mnemonic names for them) but do not affect the embedding expression evaluation.

The expression $[\mathcal{L}.@ \uparrow]$.check-uniq is evaluated to yield the attributes named `check-uniq` in clusters adjacent through D-edges from the @ attribute vertex of the goal cluster. Starting from $\mathcal{L}$ (the goal symbol vertex) we "narrow" consideration to the cluster's @ vertex, then follow all D-edges, then "widen" consideration to the cluster centers at which we have arrived. Finally we narrow back to the `check-uniq` attribute vertices of all of these clusters.

Finally we explain the expression

$$f([\mathcal{L}.@ \uparrow].e, \lambda x.x \downarrow val \cap imports \neq \emptyset)$$

This must refer to all export lists of modules that we import from. $\mathcal{L}.@ \uparrow$ gets to the name attribute of all modules (except the one currently being created). We widen to get to the modules' centers and narrow to focus on their export lists. Then we filter out those that are not exporting anything that we want to import, by comparing their attribute values to parameters given the production. The attribute vertices that remain are precisely those export lists to which we must embed.

Embedding of the daughter graph is specified by including special vertices called *sockets* in the bodygraph (denoted by retangles in the Module Interconnection example productions). Sockets are not instantiated into vertices to be added as part of the daughter graph into the host graph. Rather they are labeled by Edge-End Embedding Expressions and represent each vertex in the goal cluster's neighborhood to which their label evaluates. An edge incident on a socket is instantiated into one edge for each vertex that the socket represents. These are the edges that embed the daughter graph into the host graph.

Definition 4 Bodygraph *extends the definition of graph by having a third set of vertices (sockets) and a labeling function to assign an Edge-End Embedding Expression and a rolename to each socket.*

A bodygraph is well-formed *if:*

- *The graph resulting from removing all the sockets and their incident edges is well-formed;*
- *All E^4s (i.e. socket labels) evaluate to attribute vertex sets (the typing system makes this easy to check statically);*
- *Sockets are only adjacent to attribute vertices. An implication is that there are no edges between sockets, which is necessary for being 1-NCE.*

We extend the previous definition of D-edges in bodygraphs: a D-edge is allowed to have source or target be a socket. By the third requirement, and since C-edges originate from symbol vertices, we do not have to extend that definition.

Technically, bodygraphs must also have priorities and values for each attribute vertex, but these are calculated and used dynamically, so we don't have to specify them here.

Definition 5 *A* production *is a pair $< L, R >$ where L is a symbol and R is a well-formed bodygraph.*

In the examples of section 2, productions also have parameters that allow them to customize the constant values of some of the new attributes created. Using such parameters is a shorthand for an infinite set of productions, one for each possible combination of values. This is just a way to implement the "intrinsic" attribute values [14] of classical attribute grammar theory, and is ignored in the sequel.

Definition 6 Rewrite *of a graph G by a production p starting from a symbol vertex v to produce a graph H (denoted $G \rightarrow^p_v H$) is accomplished in the following steps:*

- *Verify that the label of v equals L_p (the rewrite cannot be done otherwise);*
- *Identify the cluster C around v by following all C-edges;*
- *Evaluate each E^4 that appears as a label of a socket in R_p. The result of each is some subset of attribute vertices and we further restrict these to the neighborhood of C;*
- *Produce an intermediate graph $I = (G - C) + (R_p - S)$ (where S is the sockets of R_p). I is the host graph plus an instantiation of the bodygraph;*
- *For each edge from a vertex w in R_p to a socket s in R_p, placed edges from the vertex in I corresponding to w to each vertex in I corresponding to vertices in G that were identified by evaluating s's label;*
- *Repeat this for edges that go the other way between sockets and other vertices in the bodygraph;*
- *Call the result (of adding the embedding to I) H.*

Theorem 7 *Within a given world, rewrites preserve graph well-formedness.*

Proof is tedious but straightforward. Since a production's bodygraph is well-formed it only allows placement of new vertices and edges into the host graph so that the result satisfies the well-formedness criteria.

Embedding during rewrites is restricted to the 1-neighborhood. When symbols act as abstractions for complex computations to be expanded later, it may be useful to have placeholding adjacencies so that needed attributes are in their neighborhood to be embedding to when expansion occurs. This is the purpose of the `@` attribute of the module interconnection example. It has edges from each name so that when `eGoal` is rewritten to create a new module, the new module can get correct import and export adjacencies.

3.3 Obligations and Roles

Consider an edge in a graph between two attributes. The attribute vertex at the source of the edge is said to have an *obligation* to the attribute at the target of the edge, *viz.* to produce information which the target can consume. Each attribute vertex therefore has an *obligation set*, its immediate successors in the graph, and a *resource* set, its immediate predecessors in the graph. When the graph is rewritten, in order for the rewrite to be meaningful it is necessary that the obligation/resource relations from the cluster being rewritten to its neighborhood are satisfied after the embedding of the new daughter-graph. In other words, if v is a vertex in a cluster about to be rewritten, w is a vertex in the neighborhood of the cluster, and v has an obligation to w, then after the rewrite a new vertex z must have been introduced into the graph such that z now has an obligation to w.

In practice, obligation/resource relations may be very complex. For example, in our module interconnection example, an `e` vertex has an obligation to an indeterminate number of `check-im` vertices, which depends on what variables the other modules actually import. Other operations may require a specific number of inputs. We therefore characterize operations as requiring an exact number of inputs, 0 or more inputs (as is the case in our example), or 1 or more inputs. A purpose of the static analysis algorithms in appendix A is to check that rewriting maintains obligation/resource relations.

To be able to perform such checks, we introduce the idea of *roles*. Roles are values from a preselected set of rolenames, and are used to annotate all attribute and embedding expression vertices in the grammar according to their *purpose*. In our example, roles are $R_0, R_1, \ldots$. (Space on the diagrams did not permit more mnemonic role names). An n attribute vertex has role R_0, and any embedding expression for such a vertex must also have role R_0. We can think of R_0 as "the obligation to supply a name". `check-uniq` vertices have role R_1; this role may be called "the possibility to check names". We want all vertices with role R_0 to

be connected to vertices with role R_1. Roles help in grammar engineering by forcing the grammar-writer to focus clearly on such issues.

When performing embeddings, roles must match, *i.e.* if an embedding expression labeling a socket with role r evaluates to a vertex v in the graph, then v must have role r. This is the key to our grammar analysis; we know when analyzing the grammar that only vertices with appropriate roles can be possible connection points, and then check that all possible connection points satisfy the obligations. Further analysis to check that attribution rules get the correct numbers of inputs can then be checked in a secondary phase.

3.4 Attributed Graph Grammars

Definition 8 *A* grammar *is a triple* $R = < W, P, Z >$ *where:*

- W *is a world;*
- P *is a finite set of productions with respect to* W*; and*
- $Z \in sym_W$ *such that* $A_W(Z) = \emptyset$ *(i.e.* Z *has no attribute names).*

A single vertex labeled by Z_R is a well-formed graph. We call this the *axiom* (graph) of R. A *sentence* with respect to a grammar R is a graph that can be derived through a finite number of rewrites starting from the axiom and using productions of R.

In the next section we define some properties to guarantee attribute evaluability and facilitate efficient incremental change propagation. We also give some conditions and results through which static grammar analysis may check for satisfaction of these properties.

4 Grammar and Graph Properties

Recall from definition 2 that every attribute vertex has a priority, an attribution rule, and a value.

Definition 9 *A graph is called* prioritized *if every topological sort of its attribute vertices produces a sequence of their priorities which is non-decreasing.*

Definition 10 *An attribute vertex is called* evaluable *if D-edges exist to provide argument values to each parameter of its attribution rule and if those arguments are of appropriate types.*

A graph is evaluable *if every attribute vertex in it is evaluable.*

Definition 11 *An attribute vertex is called* consistent *if it is evaluable and its value is equal the the result of evaluating its attribution rule.*

A graph is consistent *if every attribute vertex in it is consistent.*

We now give conditions sufficient to establish that a grammar produces only prioritized and evaluable graphs and show how attribute vertex evaluation can be done efficiently on a prioritized evaluable graph in order to make it consistent.

4.1 Prioritizing Graphs

Method 12 *The static analysis algorithm in appendix A will either fail (indicating that there is a potential cyclic dependency among the attributes of a bodygraph), or return a partial ordering on the attribute and socket vertices for each bodygraph, such that the ordering captures all possible transitive dependencies among attribute vertices instantiated by the production. We call this ordering a* relative priority numbering.

The orderings returned by the algorithm are a conservative approximation of the transitive dependencies that may arise in practice. The algorithm is a generalization of Kastens analysis algorithm for attributed string grammars [10].

Method 13 (Priority number interpolation) *When a production p is used to rewrite a cluster C, examine the priority numbers of the attribute vertices in the neighborhood of C that are represented by sockets of R_p. For every socket s determine the maximum and minimum of all priorities of vertices represented by s. In practice we examine the bodygraph to decide for each socket whether we need the maximum or minimum (or both) and only find what is needed. This saves some work, but does not change the time complexity. Call these rational numbers* $\max(s)$ *and* $\min(s)$.

If there are sockets s and t where the $priority_s < priority_t$ *and* $\max(s) \not< \min(t)$ *then the algorithm fails.*

For every attribute vertex v instantiated by the rewrite, assign a priority number n such that, if there is a transitive dependence (as determined by method 12 from socket s then $n > \max(s)$ *and if there is a transitive dependence to socket t then* $n < \min(t)$ *(Because we use rational numbers this is always possible; see [1]). n must also be consistent with the assignment of priorities to other instantiated attribute vertices according to the relative priorities computed for the bodygraph. This condition can be met by simply interpolating numbers to vertices in the order identified by method 12.*

This method describes how to dynamically assign priority numbers consistent with the relative priority numbers of the bodygraph and the dependencies generated by the embedding.

Lemma 14 *Given that a grammar R completes static analysis (method 12) priority number interpolation will not fail when rewriting a sentence by a production of R.*

The static analysis algorithm predicts all possible cases where a rewrite may introduce a transitive dependency between two already existing attribute vertices. When static analysis succeeds the relative priority numbering fixed for each production's bodygraph encodes not only those dependencies that are used for the given rewrite, but those dependencies that may be created in any possible recursive rewriting of clusters introduced. Therefore, even if a production is introducing new transitive dependencies, this eventuality was prepared for earlier so that interpolation is possible (i.e. the actual priority numbers on both sides of the new transitive dependence have the correct sense of inequality).

Since the introduction of new transitive dependencies is the only situation that can make interpolation fail and static analysis (if it succeeds) covers this case, we guarantee that interpolation is always possible.

Lemma 15 *Method 13 preserves the prioritized property in graphs.*

According to the method new priorities are interpolated between attribute vertex priorities in the neighborhood of the rewrite. Also assignment follows the dependencies internal to the daughter graph that is instantiated. If the original graph was prioritized and the method succeeds, then the result will be prioritized.

Theorem 16 *Given a grammar that passes static analysis and applying method 13 uniformly (i.e. for every rewrite starting from the axiom) every graph produced is prioritized.*

Proof is by induction using lemmas 14 and 15 and noting that the axiom graph is prioritized trivially.

As a result of this theorem we assume in the sequel that all graphs are prioritized. The cost of keeping graphs prioritized is of the same time complexity as the cost of generating the graphs.

Theorem 17 *Method 13 is optimal.*

The cost of priority number interpolation is linear in the number of attribute vertices added and the size of the goal cluster's neighborhood. Rewriting requires this much work anyway. As E^4s are evaluated to consider subsets of the neighborhood, min and max information is gathered with a constant factor additional time. Similarly, the relative priority numbers of the bodygraph give us an efficient precomputed order in which to instantiate attribute vertices and give them priority numbers.

4.2 Attribute Evaluation

We use the standard definition of attribute evaluation, as found, for example, in [16]. After some change to a graph, optimal incremental change propagation is performed guided by static analysis. This will work whenever all attribute vertices *evaluable*, as defined above.

Requirement 18 *When an attribute vertex is instantiated from the bodygraph of some production by a rewrite, it must immediately be evaluable.*

This constrains static analysis to check that each attribution rule gets arguments either locally from the production's bodygraph or remotely from a socket. If from a socket, static analysis must guarantee that the socket label will always evaluate to a set able to satisfy the attribution rule.

Requirement 19 *Every rewrite must fulfill the obligations of the goal cluster, i.e. every D-edge target outside the cluster (where the edge originated from some attribute vertex of the cluster) is a part of the set that some E^4 evaluates to. The socket labeled by that E^4 must have a D-edge in-incident on it.*

Again we use static analysis to check whether a grammar will always satisfy this run-time requirement.

Theorem 20 *Any graph resulting from a grammar satisfying the previous two requirements is evaluable.*

Proof is by induction on rewrites. After each rewrite requirement 18 guarantees that newly created attributes are evaluable and requirement 19 guarantees that attributes in the neighborhood remain evaluable.

Method 21 (Attribute change propagation) *Given a prioritized evaluable graph, incrementally reestablishing consistency from an arbitrary initial working-set p of attributes[2] that are known to need evaluation proceeds by:*

- *Place elements of p into a priority queue q according to their priority numbers;*
- *Repeat the following until q is empty—*
- *Remove the front of q and evaluate it, putting the result as the value of the attribute vertex;*
- *If the value of the vertex changed, add all its graph successors to q (inserted according to their priority numbers).*

After a rewrite the attribute change propagation algorithm is run. Forming the initial priority queue is linear time since the attribute vertices' priorities are established from relative priority numbers so their sorted order is precomputed by static analysis. A variant of the model is to allow multiple rewrites between attribute reevaluation, for efficiency. In this case forming the queue is by merging sequences rather than sorting all the attribute vertices.

The cost of method 21 is computed as follows. If *Influenced* is the set of attributes reevaluated by method 21, then the number of evaluations is | *Influenced* |. Each evaluation is followed by the cost of an

[2]For this paper, p is just the attribute vertices instantiated from the bodygraph of the production used in the rewrite.

update to the priority queue. Clearly, the queue can be no larger than the size of the *Influenced* set, so the worst-case complexity of the algorithm is $O(|\mathit{Influenced}| \cdot \log |\mathit{Influenced}|$. In practice the size of the priority queue will usually be significantly smaller than $|\mathit{Influenced}|$, especially for sparse graphs, so the algorithm will approach linear-time behavior. Note that because of the flexibility of the embeddings, we cannot in general bound the out-degree of any attribute vertex, so, unlike the standard evaluation algorithm on trees [14] the cost of maintaining the worklist of attributes needing reevaluation is not bound by a constant.

References

[1] B. Alpern, A. Carle, B. Rosen, P. Sweeney, and K. Zadeck. *Incremental Evaluation of Attributed Graphs.* Technical Report CS-87-29, Department of Computer Science, Brown University, December 1987.

[2] Veronique Donzeau-Gouge, Gilles Kahn, Bernard Lang, and Bertrand Melese. Documents structure and modularity in mentor. In *Proceedings of the ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 141–148, Pittsburgh, Pa, May 1984.

[3] Gregor Engels and Wilhelm Schafer. Graph grammar engineering: a method used for the development of an integrated programming environment. In Hartmut Ehrig, Christiane Floyd, Maurice Nivat, and James Thatcher, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT), LNCS 186*, pages 179–193, Springer-Verlag, 1985.

[4] Herbert Gottler. Attributed graph grammars for graphics. In Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, editors, *Proceedings of the second International Workshop on Graph Grammars and their Application to Computer Science, LNCS 153*, pages 130–142, Springer-Verlag, 1982.

[5] Herbert Gottler. Graph grammars and diagram editing. In G. Rozenberg H. Ehrig, M. Nagl and A. Rosenfeld, editors, *Proceedings of the Third International Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 216–231, Springer-Verlag, Heidelberg, 1987.

[6] D. Janssens and G. Rozenberg. Graph grammars with neighbourhood-controlled embeddings. *Theoretical Computer Science*, 21:55–74, 1982.

[7] Simon M. Kaplan. *Incremental Attribute Evaluation on Graphs (Revised Version).* Technical Report UIUC-DCS-86-1309, University of Illinois at Urbana-Champaign, December 1986.

[8] Simon M. Kaplan and Roy H. Campbell. Designing and prototyping in grads. In *Proceedings of the Second IEE/BCS Conference on Software Engineering, Conference Publication 290*, pages 55–59, IEE, Liverpool, July 1988.

[9] Simon M. Kaplan, Steven K. Goering, and Roy H. Campbell. Supporting the software development process with attributed nlc graph grammars. In G. Rozenberg H. Ehrig, M. Nagl and A. Rosenfeld, editors, *Proceedings of the Third International Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 309–325, Springer-Verlag, Heidelberg, 1987.

[10] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

[11] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[12] Manfred Nagl. A software development environment based on graph technology. In G. Rozenberg H. Ehrig, M. Nagl and A. Rosenfeld, editors, *Proceedings of the Third International Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, pages 458–478, Springer-Verlag, Heidelberg, 1987.

[13] David Notkin. The gandalf project. *Journal of Systems and Software*, 5(2):91–106, May 1985.

[14] Thomas Reps. *Generating Language-Based Environments*. MIT Press, 1984.

[15] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

[16] William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, New York, 1984.

A Static Grammar Analysis

There are two reasons for analyzing a grammar:

- To determine an approximation of transitive dependencies that may occur in sentences derived by the grammar (that includes at least those dependencies that will arise in practice). We do this in support of method 12 to assign relative priority numbers; and
- To verify that rewrites preserve evaluability (i.e. to show that requirements 18 and 19 hold).

These two goals are unified through the concept of role. By giving each vertex and each socket a role the analysis algorithm has more information[3], so that it can get a tighter approximation of transitive dependencies. This reduces (and in practice eliminates) the problem of inducing cycles of dependencies where no such cycles exist in practice. Also, specifying roles for attribute and socket vertices gives a handle on obligations and resources for each attribute.

Roles are not attribute names. Attribute names are assigned on the basis of type information. Roles are given to declare intended use for the attribute values.

A.1 Inducing Transitive Dependency Information

Definition 22 *A socket* matches *an attribute vertex if and only if their roles are equal.*

This is a conservative approximation of sockets to attribute vertices because at run time the socket role must equal the attribute vertex's role and its E^4 must evaluate to include that attribute vertex for it to represent the vertex.

It is possible to generalize roles to *role sets*, so that an attribute can have multiple roles, which is often useful in practice. This generalization is straightforward and is not considered in this paper.

Method 23 *Let $B_1, B_2, \ldots, B_n$ be the production bodygraphs of a grammar, R. Let $I_1, I_2, \ldots, I_n$ be* induced dependency bodygraphs *with values as assigned during the algorithm. Let D be a set of triples where the first element is a symbol and the other two elements are socket vertices of a bodygraph that is the right-hand-side*

[3] In an earlier work [7], embedding expressions were simply (nonterminal) symbols and there were no roles; attempting to do static analysis in this model often induced so many false cycles that the class of grammars which passed the analysis was too restricted. The approach presented in this paper eliminates this problem, by using roles, powerful embedding expressions, and embedding to attributes directly rather than to their associated terminal or nonterminal symbol.

of a production with the symbol as goal. D will be built by the algorithm as dependencies across sockets within each bodygraph are discovered.

Initialize each I_i to `transitive-closure(` B_i `)`.
For each edge $(s_1, s_2) \in I_i$ between sockets, add the triple
    $< L_{p_i}, s_1, s_2 >$ *to D, where L_{p_i} is the goal of production i.*
While D continues growing do
  For each I_i do
    For each symbol vertex $v \in I_i$, where $sl(v) =$ 'X' do
      For each triple $< X, a, b > \in D$ do
        If there exists vertices y, z in the cluster neighborhood of v
          and y matches a and z matches b and $(y, z) \notin I_i$
          then add (y, z) to I_i.
    If any new edges were added to I_i do
      Let $I_i =$`transitive-closure(`I_i`)`
      If I_i is cyclic then the algorithm fails
      Add newly induced socket dependencies $< L_{p_i}, s_1, s_2 >$ to D.

The result is a set of graphs $I_1, I_2, \ldots, I_n$ with the same vertices as the bodygraphs and a superset of the transitive closures of the edges.

This algorithm is a generalization of the first phase of Kastens algorithm for analyzing attributed string grammars to test if they are *ordered* [10]. As with Kastens algorithm, experience thus far suggests that "real" grammars which are actually acyclic pass the test.

Given an acyclic graph I_i for production p_i, we assign relative priority numbers to attribute vertices and sockets of B_i. Priority interpolation (method 13) requires that numbers assigned to sockets be equal unless there is a dependence induced between them. Since I_i is transitively closed, it is trivial to assign numbers 0, 1, *etc.* so that both this, and monotonicity of numbering across dependencies are preserved.

A.2 Deriving Obligation Information

Static analysis can determine if a vertex meets its obligations, and whether the resources it needs are supplied to it, in the face of all possible rewrites on the graph. Passing this analysis guarantees that requirements 18 and 19 are met for the grammar.

Checking these requirements involves an inductive algorithm. First the conditions are shown to hold for rewrite of the initial graph (the base case), and then for any other rewrite. In both cases, the methodology is the same: For each attribute vertex v with role r, consider all the sockets adjacent to it. Suppose that s is the role of the socket. Identify all attribute vertices with role s. Now check that the rewrite of the cluster associated with each vertex will place an edge to vertices with role r. Finally, check that all rewrites of v will introduce a vertex with role r connected via a socket to vertices with role s.

Note that since roles are designed to correspond to the *purpose* of an attribute, there should be a close correlation between the vertices identified in the algorithm as potential connection points, and vertices to which connections are actually made. Roles therefore act to reduce the number of possible connection points. In practice we find this correlation to be a 1-to-1 relation; if this is not true, then the grammar is probably improperly engineered.