Proc. TAGT'98 - Theory and Applications of Graph Transformations, Paderborn, Germany, November 1998

GXL - A Graph Transformation Language with Scoping and Graph Parameters

Medha Shukla Sarkar
Dorothea Blostein
James R. Cordy

Department of Computing and Information Science
Queen’s University, Kingston, Canada K7L 3N6

shukla@qucis.queensu.ca, blostein@qucis.queensu.ca, cordy@qucis.queensu.ca

Abstract

GXL - the graph transformation language presented in this abstract is a programmable graph rewriting language that combines ideas from current tree rewriting technology, exemplified by the TXL language, and general graph rewriting systems. From TXL it inherits the idea of scoping (limitation of a transformation rule to a particular subtree or subgraph) and parameterization (working with multiple independent copies of a subtree or subgraph). From graph rewriting it inherits the generalization to graphs from trees. The resulting language addresses both problems in tree rewriting imposed by the limitation to trees, and problems in graph rewriting imposed by the lack of scoping and parameterization.

1. Motivation

Graph rewriting systems such as PROGRES [1] have proven very powerful and expressive notations for dealing with large graphs, but can be awkward or inefficient when specifying complex graph transformations that involve rules to be limited to particular subgraphs of the host graph, or to deal with multiple copies or views of a matched subgraph. Both of these problems are well solved in the tree rewriting community by borrowing ideas from pure functional programming [2] and rule based programming [3].

Tree rewriting languages such as TXL [4], on the other hand, excel at implementing complex transformations involving multiple copies and views of matched subtrees and application of different rulesets to different matched subtrees, at speeds orders of magnitude faster than the fastest graph rewriting systems. But the limitation to trees can make it awkward or inefficient in TXL to implement transformations such as call graph closure that are more naturally expressed using general graph rewriting.

GXL, the graph transformation language presented here, is a programmable graph rewriting language that aims to address the limitations of both graph rewriting and tree rewriting by a synthesis of the two that uses the strengths of each one to address the weaknesses of the other. From TXL, we borrow the first-order functional programming constructs that give us scoping and parameterization. From graph rewriting systems, we borrow the ability to deal with large general graphs.

2. Design of GXL

The design of GXL is greatly influenced by the tree-transformation language TXL [4]. In TXL, scoping limits the scope of the application of a rewrite rule to particular subtrees. It is possible to identify all subtrees matching a certain pattern and it is then possible to apply a specific set of rewrite rule(s) to only those subtrees. For example, in the parse tree of a PL/I program we can identify by pattern only those subtrees representing, say, procedures that call the procedure P, and apply subrules that add initialization and finalization statements for P's global variables to the beginning and end of those procedures (Figure 1).

```
rule addInitializersAndFinalizersToProceduresCallingP
    % Find each [procedure] sub-parse tree in the program parse tree ...
    replace [procedure]
            Proc [procedure]
    % ... that contains a [statement] sub-parse tree calling the procedure with name 'P' ...
    deconstruct * [statement] Proc
            CALL P ;
    % ... and that does not already contain a [statement] sub-parse tree referencing 'P_initialize' ...
    deconstruct not * [statement] Proc
            CALL P_Initialize ;
    % ... and apply the subrule [addInitializeFinalizeForP] only to it
    by
            Proc [addInitializeFinalizeForP]
end rule

function addInitializeFinalizeForP
    % Find the [statement*] sub-parse tree (i.e., all the statements) in the scope (i.e. the procedure)
    replace * [statement*]
            Statements [statement*]
    % Construct a new [statement*] sub-parse tree to call 'P_Initialize' ...
    construct CallInitializer [statement*]
            CALL P_Initialize ;
    % ... and one to call 'P_Finalize' ...
    construct CallFinalizer [statement*]
            CALL P_Finalize ;
    % ... and add them to the result ( the built-in [.] subrule attaches subtrees to a tree )
    by
            CallInitializer [. Statements] [. CallFinalizer]
end function
```

Figure 1. TXL tree transformation rules to add [call_statement] subtrees for calls to 'P_initialize' and 'P_finalize' to each subtree labelled [procedure] that contains a subtree labelled [call_statement] that references the procedure 'P'

TXL tree patterns capture and name subtrees of the trees that are matched by the pattern. These names can then be used to refer to the captured subtrees as scopes and parameters of other rewrite rules. TXL uses value semantics for the subtrees, which means it can easily deal with multiple copies of subtrees, including passing them as parameters to subrules. TXL's optimizer uses a very efficient implementation for this which minimizes subtree copying. We hope to carry over this efficiency to GXL.

TXL has a well developed semantic model [5] and is an expressive, strongly typed programming language which supports transformational programming using tree rewriting. The adaptation of TXL ideas to graph rewriting in GXL is expected to result in a similarly expressive graph transformation language.

3. GXL Notation

In general, GXL adopts the syntax, language features and style of TXL, adapted and extended to handle general directed graphs. The scope of a GXL rule is a general directed graph; patterns and replacements (left- and right-hand sides respectively) in GXL are expressed in terms of subgraphs, nodes and edges, using TXL's pattern binding concept to attach names to matched items.

Figure 2 gives a trivial example of a graph rewriting production in GXL notation. The example is a simple rule to add reverse edges labelled 'calledby' to every pair of nodes linked by a 'calls' edge in the host graph. Ideally, patterns and replacements will be expressed graphically in a production implementation of GXL. In the prototype implementation, textual approximations using ----> to represent edges are used.

The following sections focus on the details of the two main features of GXL that distinguish it from other graph rewriting systems - scoping and parameterization, and give two examples of the use of these features in real graph rewriting problems.

```
rule addCalledByEdges
    % Find each 'calls' edge ...
    replace $ [subgraph]
          P [node]  -- calls -->  Q [node]
    % ... and add a 'calledby' edge
    by
          P   -- calls -->  <-- calledby --  Q
end  rule
```

Figure 2. Trivial example of GXL syntax. Embedding is handled by treating bound nodes and edges as gluing points - in this case nodes bound to P and Q retain their embedding in the result. See section 4 for details.

4. GXL - Scoping

In GXL, "scoping" means that a graph production can be applied to only some subpart of the host graph. Existing formulations for graph rewriting typically assume that the entire host graph is available for rewriting. Scoping provides the programmer more control over the situation. One graph production can define a scope that is used by other graph productions. In GXL, a scope can be a subgraph of any graph and multiple subgraphs can be handled simultaneously, i.e., graph productions can be applied to each and the result reconstructed from them.

In GXL, the notion of host graph is defined by the main scope of the application, that is, there is a "main" graph production which is the only one applied to the host graph, and other productions are applied to subgraphs matched by the main production. In turn, the productions applied to these subgraphs match sub-subgraphs to which other productions are applied, and so on, in functional programming style.

We refer to subgraphs as subscopes within the main scope (i.e., the host graph). The LHS of a graph production in GXL is a subgraph pattern that is to be matched within the current scope of the production. The RHS of a graph production gives the subgraph replacement for the pattern matched in the current scope. When a scope is defined in GXL (i.e., a subgraph of the main scope), it can be refined to subscope (i.e., a sub-subgraph of the main scope) and so on using pattern refinements ('deconstruct' phrases).

Like TXL, GXL uses value semantics and hence does not work with a single entity that is the host graph, as in other graph rewriting systems, but rather with any number of graphs and subgraphs captured by name in productions and copied when necessary. The algebraic graph rewriting model is used for embedding. That is, we use gluing nodes in the graph production to show how the replacement (RHS) subgraph will be embedded in the host graph. This model fits naturally into the TXL paradigm of binding nodes and subgraphs to names in the pattern (LHS) and using these names to embed them in the replacement (RHS).

5. GXL - Parameterized Rules

GXL also adapts from TXL the notion of "Graph parameters" that allows subgraphs to be passed as parameters to a graph production. This allows more flexible processing than is permitted by hard- coding the graphs in the left- and right-hand sides of productions, as is done in other graph rewriting systems. Passing graphs as parameters allows complex productions to be applied, involving many independent subgraphs at once. Since GXL uses value semantics, graph parameters that represent subscopes of the main scope can be passed to graph productions without fear of unintentionally changing or interfering with the main scope (host graph).

In order to retain TXL's high degree of efficiency and programmability, graph matching in GXL is deterministic. In most graph rewriting languages, if there are several possible matches for the LHS of a rule, then one of these matches will be picked at random. This is not the case for GXL. GXL imposes a canonical order on the nodes of the scope (host graph), and then searches for subgraph matches according to this order. This deterministic approach is necessary to allow the scoping feature in a graph transformation language to have a well defined semantics. GXL searches for pattern matches in the scope graph of a production in a strict order. In other words traversals on graphs in GXL are carried out in a strict order on the nodes. By contrast, a nondeterministic approach does not impose any strict order of traversal (matching) on the graph (scope) and as a result, it is impossible to unambiguously break down the main scope into subscopes for further refinement.

6. Examples Using Scoping and Parameters in GXL

To illustrate the advantages of scoping and graph parameters in graph rewriting we show the following examples. The first example (Figure 3) illustrates the use of GXL scoping to identify neighborhood subgraphs of nodes with a particular label. This has applications in world wide web searching, where we are interested in finding documents containing certain text, within the link neighborhood of a known document. In this example, we are interested in all the pages mentioning Rewriting Systems within the neighborhood of a page mentioning Software Engineering. Here we define neighborhood as a page reachable by traversing at most 2 links from page Software Engineering. This example is illustrative of the kinds of computations that GXL can solve conveniently and expressively.

In a standard graph transformation language, this would need a complex graph production. For example, first the node representing page Software Engineering would have to be matched. Then a mark would have to be made to the neighborhood of page Software Engineering by setting a special flag attribute for nodes in the neighborhood. Then we could apply a rule that matches nodes that are both marked and contain references to Rewriting Systems.

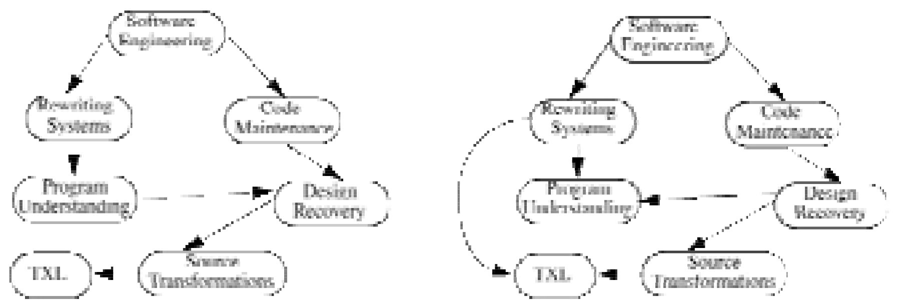
GXL scoping constructs allow this type of graph production to be carried out conveniently. One rule matches the subgraph which is the "neighborhood" of the Software Engineering home page. Then another rule or a set of rules match pages containing references to Rewriting Systems within the scope.

```
function main
    replace [graph]
          MainGraph [graph]
    by
          MainGraph [addTxlLinkInNeighborhoodOfSoftwareEng]
end  function
```

```
rule addTxlLinkInNeighborhoodOfSoftwareEng
    replace [subgraph]
        % Match the neighborhood of 'Software_Eng' to distance 2
        node ’Software_Eng’ --link--> First_Nodes [node*] --link--> Second_Nodes [node*]
    construct Neighborhood [subgraph]
        % Gather the entire neighborhood into a single scope
        node ’Software_Eng’ --link--> First_Nodes --link--> Second_Nodes
    by
        % Add a link to GXL to nodes with label ‘Rewriting Systems’ in that neighborhood
        Neighborhood [addTxlLink]
end rule

rule addTxlLink
    replace [subgraph]
        node ‘Rewriting Systems’
    by
        node ‘Rewriting Systems’ --link--> node ‘TXL’
end rule
```

Figure 3. This GXL program finds all nodes named Rewriting Systems in the link neighborhood of nodes named Software Engineering. It then updates these nodes by adding a link to TXL. When this program is applied to the graph in Figure 4(a), the graph in Figure 4(b) results.



(a) (b)

Figure 4. An example application of the rewrite rule given in Figure 3 to a small host graph. This example has applications in world wide web searching for particular links to home pages in a specific neighborhood.

The second example illustrates the use of GXL scoping and parameterization for inlining called functions in a program call graph (Figure 5). We have a labelled graph representing the design structure of a computer program. In essence, the graph is an entity-relationship (ER) diagram with nodes representing entities such as procedures, functions, variables, etc. and arcs representing the relationships between the entities, such as "calls", "contains", etc. To represent the inlining of a function we do the following: the function F itself is represented as a subgraph with a "function" arc pointing to itself, and zero or more "contains" arcs pointing to the entities and relationships about the body of the function. A call to the function is represented as a "calls" arc from the calling entity (e.g., procedure P) to the F node.

The transform written in GXL is: one rule finds each function subgraph (call it G) and, another rule takes each found function subgraph (e.g., G) as parameter and replaces every subgraph that calls G by a new subgraph that replaces the call by a copy of the "contains" subgraph of G. That is, the transform inlines the subgraphs for the bodies of all functions, replacing function calls with their body subgraph.

```
function main
    replace [graph]
          MainGraph [graph]
    by
          MainGraph [inlineAllFunctionCalls]
end  function

rule inlineAllFunctionCalls
    replace [subgraph*]
          HGraph [subgraph*]
    deconstruct * HGraph [subgraph]
          % find that pattern which has a function contains subgraph
          function F [id]  --contains-->  Subgraphs [subgraph*]
          % we want the contains subgraph of the function F to be passed
    by
          HGraph   [ inlineFunctionCall  Subgraphs  F ]
end  rule

rule inlineFunctionCall  Function_Subgraph [subgraph*]  Func_Id [id]
    replace [subgraph]
          procedure P [id]  --calls-->  function Func_Id
    by
          procedure P  --contains-->  Function_Subgraph
end  rule
```

Figure 5. GXL program to inline function calls in a software design graph. An application of this rule is illustrated in Figure 6. Note that this source code need not be altered at all to adapt to another host graph representing any other software system design.

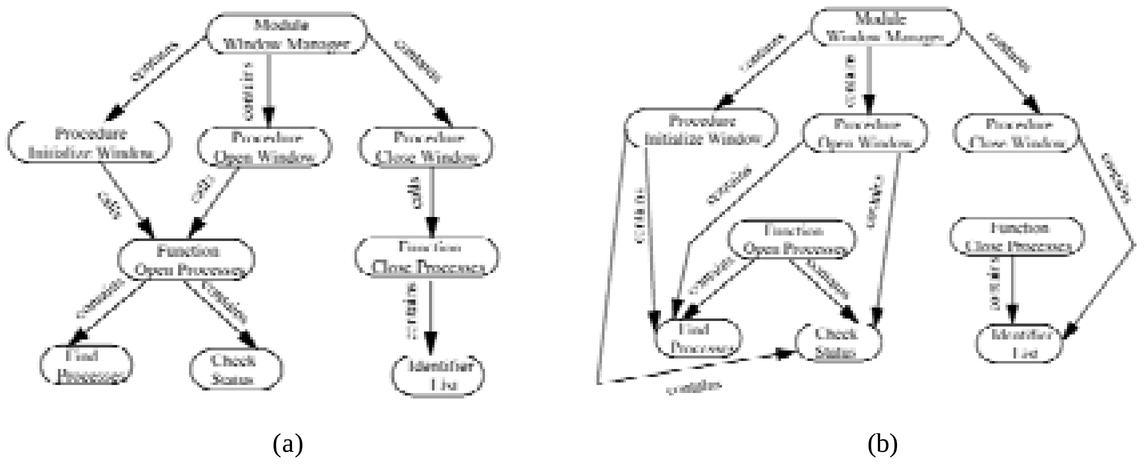


Figure 6. An example application of the GXL program given in Figure 5 to a small host graph. This example has applications in software analysis and re-engineering. The example host graph (a) is transformed to the target graph (b) by the GXL program.

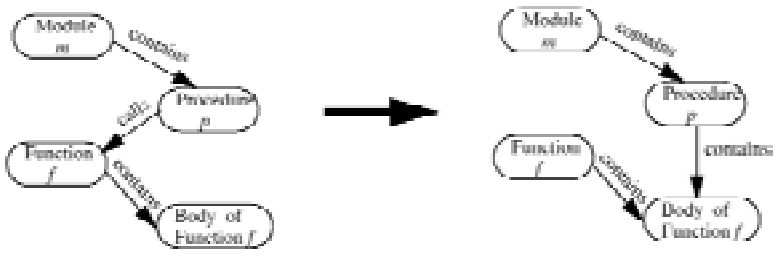


Figure 7. Graphical representation of the rewrite rule implemented by the GXL program given in Figure 5. In the long run, it is hoped that a similar graphical notation can be used to directly specify the patterns and replacements in GXL rewrite rules.

7. Conclusion

GXL is a new and different kind of graph rewriting system. Using the programming concepts and value semantics of functional programming languages, and the concepts of scoping and parameterization borrowed from the TXL tree rewriting paradigm, we hope to achieve a much more expressive and programmable graph rewriting paradigm that has been possible in existing graph grammar systems. It is also hoped that some of the high efficiency of tree rewriting systems can be adapted into GXL.

A prototype implementation of GXL is under development, is already able to run simple programs. The present version of the prototype uses a textual representation for input graphs. in the long term, we will be using graphs as diagrams directly in the language as is shown in Figure 7.

A formal programming language semantics for GXL is under development, using the existing denotational semantics for the TXL language as a basis and extending to include semantics for subgraph embedding and deterministic graph search. Neither of these problems is a difficulty for tree rewriting.

Work exploiting the TXL paradigm for handling graphs is also underway at the University of Aachen in Germany, and in industry at Legasys Corporation in Canada.

Acknowledgements

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] A. Schürr, A. Winter and A. Zündorf, "Visual Programming with Graph Rewriting Systems", Proc. 11th IEEE Symposium on Visual Languages, Darmstadt, Germany, pp. 326-333 (September 1995).

[2] Richard Bird, Introduction to Functional Programming Using Haskell, 2nd edition, Prentice Hall Europe (1998).

[3] W.S. Clocksin and C.S. Mellish, Programming in PROLOG, 4th edition, Springer Verlag (1994).

[4] James R. Cordy, Charles D. Halpern and Eric Promislow, "TXL: a rapid prototyping system for programming language dialects", Computer Languages 16(1) pp. 97-107 (January 1991).

[5] Andrew J. Malton, "The Denotational Semantics of a Functional Tree-Manipulation Language", Computer Languages 19(3) pp. 157-168 (July 1993).