Towards Generation of High-performance Transformations

Attila Vizhanyo, Aditya Agrawal, Feng Shi

Institute for Software Integrated Systems, Vanderbilt University
Nashville, TN 37235, USA
{viza, aditya, fengshi }@isis.vanderbilt.edu

Abstract. In this paper we introduce a graph rewriting language, called Graph Rewriting and Transformation (GReAT), and a code generator tool, which together provide a programming framework for the specification and efficient realization of graph rewriting systems. We argue that the performance problems frequently associated with the implementation of the transformation can be significantly reduced by adopting language and algorithmic optimizations and partial evaluation.

1 Introduction

The Model Driven Architecture (MDA) [3] advocates the use of models in software development through either Unified Modeling Language (UML) [2] or though domain specific languages supported by Meta Object Facility (MOF) [4]. In the later approach, transformations are to bridge the semantic gap between domain specific models and implementation. Other software engineering areas such as Model-Integrated Computing [1] and tool integration also have a requirement of model transformations that bridge semantic gaps between design tools.

Such transformations can be formally specified and automatically implemented using Graph Rewriting/Transformation (GRT) languages [5]. A GRT language typically consists of transformation rules where a pattern graph is matched in the host graph and replaced with a replacement graph. The time complexity of such transformation systems is determined by (1) the sub graph isomorphism algorithm, known to be NP complete and (2) the algorithm to keep track of ready to fire productions. The complexity of such transformation system becomes unacceptable for complex transformations and large graphs. We conjecture that these concerns can be addressed at various places in a transformation system ranging from the language to its implementation.

This paper presents runtime-optimization related features of Graph Rewriting and Transformation (GReAT) [12][13][14] a graphical rewriting/ transformation language and its Code Generator (CG) that is used to implement the transformations. The optimizations have been classified into two categories, (1) language and algorithmic optimization and (2) partial evaluation and implementation optimizations.

Language and algorithmic optimizations are based on language constructs that have optimized implementation algorithms. While partial evaluation and implementation optimization are performed by the Code Generator (CG) which produces code specific to a given transformation and input/output domains. The generated code provides a significant performance boost over GRE, GReAT's generic graph rewrite/transformation engine. Since the CG does not partially evaluate the transformation based on input/output graph, the generated transformation can be reused for any graph in the input/output domains. There is some overhead associated with regeneration and recompilation of the transformation code if the transformation is changed, but those transformations that are still under development can be executed and debugged using a generic rewrite/transformation engine. Once the transformation reaches a mature state, the transformation can be compiled into a high-performance executable that is capable of performing transformations on large models.

Paper organization: Section 2 reviews the area of graph grammars and transformations. Section 3 briefly describes Graph Rewriting and Transformation (GReAT) a metamodel based model-to-model transformation language and discusses language and algorithm level optimizations in GReAT. Section 4 describes GReAT's Code Generator (CG) and implementation level optimizations. Section 5 provides some experimental results comparing the runtime performance of transformations using CG and GRE. Conclusions and proposals for future research are presented in Section 6.

2 Background

2.1 Graph rewriting and Transformations

There are a variety of graph transformation techniques described in [5][6][7][8][9][10][11]. Prominent among these are node replacement grammars, hyper edge replacement grammars, single/double pushout and programmed graph replacement systems. The next few paragraphs will discuss each approach and point out some complexity issues [5].

Node replacement grammars are a class of graph grammars that are based primarily upon the replacement of nodes in a graph. The basic production of every node replacement grammar has a LHS subgraph (called mother graph) that produces an RHS subgraph (called daughter graph). Usually the LHS subgraph consists of only one node, making this class of grammars context free. The productions can be applied whenever a mother node is found in the host graph. If two productions can be applied at the same time then the order of application is non-deterministic [5].

The execution time of node replacement grammars is bounded by graph search and tracking of ready-to-execute productions. If all the subgraphs contain only one node in the mother graph then the worst case complexity of finding a ready-to-execute production is $O(n \times r)$, where n is the number of nodes in the graph and r is number of productions. Single node mother graphs are suitable for defining and parsing

graphical languages but are restrictive and not suitable for defining complex "algorithmic" transformations.

Hyperedge replacement grammars deal with the productions that replace hyper edges by subgraphs. Each production has a hyperedge on the LHS, which is replaced by a subgraph on the RHS. Hyperedge replacement by definition is confluent, associative and parallelizable. The time complexity and shortcomings are similar to the node replacement grammars [5].

Another approach to graph grammars is the algebraic one. The approach is based on a generalization of Chomsky grammars from strings to graphs. The main goal was to generalize the string concatenation to a gluing construction of graphs. The gluing of graphs is defined by algebraic constructions called pushouts. The pushout approach has been borrowed from a more general field of category theory. Significant research has been done on pushouts and how productions can be parallelized. The algebraic approach is more powerful and has concepts for sequencing and parallelizing the rules [6].

The algebraic approaches, in the general case, have subgraphs in the LHS and thus subgraph isomorphism algorithms are required find a particular LHS subgraph in the host graph. Subgraph isomorphism is known to have a order complexity of $O(n^p)$ where, n is the number of host graph nodes and p is the number of nodes in the subgraph. The time complexity of finding a ready-to-execute transformation, in the general case, is $O(r \times n^p)$ where r is the number of transformation in the system. In the algebraic approaches it is possible to specify a sequence of the transformation rules. This eliminates the need for finding the next ready-to-fire transformation. The sequencing of rules is limited only to sequential and parallel execution of the rules. It lacks high-level sequencing constructs such as conditional branching of productions, loping and recursion. The lack of high-level sequencing means that the user cannot represent and/or choose between depth-first search and breadth-first search.

The last approach to be discussed is that of programmed replacement systems, which are the most practical of all the approaches discussed so far. The leading research result is the PROgrammed GRaph REplacement System (PROGRES) [6]. The major breakthrough of PROGRES is that it concentrates equally on productions and sequencing of the productions. Thus the system has a graph replacement language that defines the productions and also programming constructs that define the order of application of the productions. The PROGRES system consists of two parts - the first is a logic based structure replacement system that describes graph transformation productions of the language, and the second is a collection of programming constructs such as recursion, non-deterministic application of productions, conditions and loops. Apart from these PROGRES can also specify static integrity constraints on the graphs. The time complexity of PROGRES based transformations is in the hands of the user [5][6][7].

3 Language and Algorithmic Optimizations

3.1 GReAT: Graph Rewriting and Transformation

Graph Rewriting and Transformation (GReAT) [12][13][14] is the transformation language developed for model-to-model transformations/rewriting. This section provides a brief overview of GReAT, while [12] provides a more detailed description. The operational semantics of GReAT is formally defined in [14]. GReAT is based on the theoretical work of graph grammars and transformations [5] and belongs to the set of practical graph transformations systems, like AGG [8] and PROGRES [7]. GReAT has two parts: (1) graph transformation language, and (2) control flow language. The graph transformation language is used to specify transformations on localized subgraphs and follows the Single Pushout (SPO) algebraic approach [5].

A production (also referred to as rule) is the basic unit of transformation and it contains a pattern graph that consists of pattern vertices and edges. Each pattern element has an attribute called role that specifies what happens during the transformation step. A pattern element can play one of three roles: Bind, Delete, CreateNew. The execution of a rule involves matching every pattern object marked either Bind or Delete. If the match is successful and an (optional) guard condition is true, then for each match the pattern objects marked Delete are deleted from the match and objects marked New are created. GReAT uses the UML [2] class diagram notation for the specification of patterns.

For example, in Fig. 1, *OrState*, *SubOrState*, *State*, *SubOrState/State* composition and *OrState/SubOrState* composition have the **Bind** role while *NewState*, OrState*NewState* composition and *State/NewState* association have the **CreateNew** role. The semantics of the rule is: find the pattern marked **Bind**, in this case the *OrState*, *SubOrState*, *State*, *SubOrState/State* composition and *OrState/SubOrState* composition pattern. Then, for every such pattern evaluate the *Guard* expression. Let the guard expression be "SubOrState.name = State.name". Thus only those matches that have this property will pass the guard and the rest will be discarded. Then create the objects marked *CreateNew*, in this case *NewState*, *OrState*/*NewState* composition and *State/NewState* association. Finally, use *AttributeMapping* to fill in the attributes of the newly created objects.

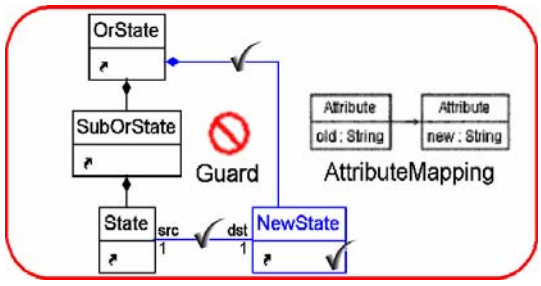


Fig. 1 An example production rule

The order complexity of the pattern matching is $O(n^p)$ and if there are many rules to choose from then the complexity of finding the correct transformation is $O(r \times n^p)$.

Traditionally, in graph grammars and transformations there is no ordering imposed on the productions, but practical model-to-model transformations often require strict control over the execution sequence. GReAT has a high-level control flow language built on top of the graph transformation language with the following constructs for improving the efficiency of the transformations: (1) pivoting and (2) sequencing.

3.2 Typed Patterns

It is well known that subgraph isomorphism is an exponential time algorithm in terms of the input graph and the pattern graph. In order to reduce the average case execution time a number of steps can be taken.

The first step is to type the pattern vertices and edges, this restrict the search to a subgraph of the host that only contains the particular types used in the pattern. If we consider a host graph having say T types of vertices and if we assume that the vertices have even distribution with respect to its type then the time complexity of matching a pattern with P_t types of vertices is $(\frac{P_t}{T} \times n)^p$. Even though the worst case execution time is $O(n^p)$ the expected case execution will have a saving. On an average, graphs contain ~30 vertex types while a pattern graph uses ~3 vertex type and thus in the average case the saving should be ~10x.

3.3 Pivoted Pattern Matching

Another optimization technique is to start the pattern matcher with an initial binding and we have named it “pivoted pattern matching”. In this technique the programmer provides an initial binding for some of the models in the pattern graph to the host graph nodes. The pattern matching is then performed in the context of the initial binding.

In Fig. 2, the pattern vertex *Pv* is initially bound to the host vertex *Hv*. This restricts the search to the area shown within dotted line. This particular optimization works well for sparsely connected graphs. For example, graph has an average connectivity, the number of edges incident on a vertex, of 3 and the greatest distance from the pivot to a vertex in the pattern graph to be 2. Then the matching algorithm will only search within a tree of depth 3 starting from the pivoted node. In general the number of host graph vertices included in the search will be c^d where c is the connectivity and d is the depth of the pattern. Hence the order complexity of the matching algorithm is $O(n^p)$, where $n = c^d$ and p is the umber of unbound vertices.

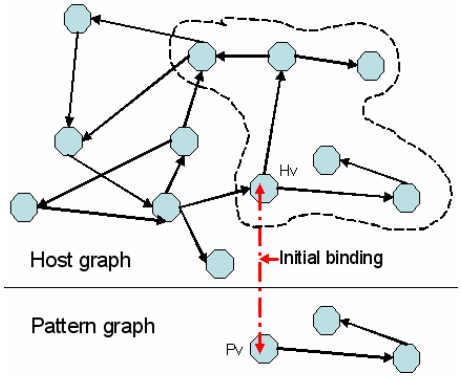


Fig. 2 Pivoted Matching

This optimization technique, when added to the typed pattern vertex technique gives a significant saving because in this case the connectivity of the restricted graph is even less. Fig. 3 shows the same rule as in Fig. 1 with the addition of *In* and *Out* ports used to provide the initial binding. The *OrState* pattern vertex is bound to a host graph vertex supplied by the port labeled *In*.

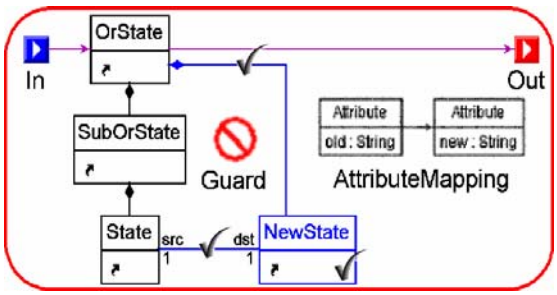


Fig. 3 Transformation Rule with pivot

3.4 Reusing Previously Matched Objects

The next optimization technique used in the GReAT is the called “Reusing previously matched objects”. The idea here is to cache previously found results and pass it on to subsequent rules as the initial binding.

For example, in Fig. 4, there are two rules, the first rule gets an input binding for *Parent* and finds all *ChildA*, *ChildB*, *Assoc* triples that correspond to the pattern. In the subsequent rule these triples are required to perform an action. Instead of finding the pattern again, the first rule passes the triples along to the next rule. For the next rule they serve as the initial binding. When a rule executes it can produce multiple matches. Each match produces a host graph object for each output port and this coherent set of objects is called a packet. These packets are sent to the subsequent rules as one unit.

GReAT supports hierarchical specification of transformation rules. High-level rules can be created by composing a sequence of primitive rules. There are two kinds of high-level rules in GReAT: *Block* and *ForBlock.* The execution semantics of the *Block* is to pass all input packets to the first contained rule, the outputs packets created by it are passed to subsequent rules and so on. After all packets have been processed and all output packets of the Block have been generated the Block returns control to its parent. Semantics for the *ForBlock* is to pass one input packet at a time through all the contained rules. After the first packet has been processed all the way to the output of the *ForBlock* the next packet is processed. These two constructs enables user to choose different traversal strategies. A *Test/Case* is also available in GReAT. It can be used to choose between different execution paths, during the transformation and is similar to 'if' statements in programming languages.

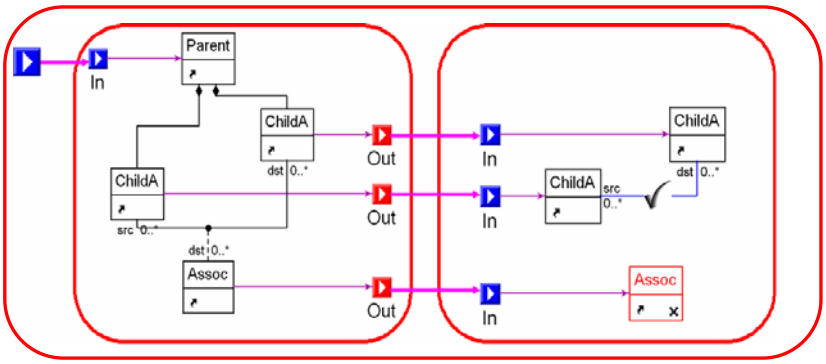


Fig. 4 Sequence of rules with passing of previous results

4 Partial Evaluation and Implementation Optimization

4.1 Motivation

In the previous section, GReAT has been introduced as a graphical rewriting language, and the language and algorithmic optimizations related to the runtime behavior of the transformations have been discussed. We further stress the importance of the execution aspects of the graph transformation system by discussing the runtime-optimization related features of the Code Generator (CG), which is used to implement the transformations.

Graph Rewrite Engine (GRE), the generic graph rewrite/transformation engine is [12] suitable for prototyping transformations but due to a high runtime overhead it is not suitable for the realization of applications with real software engineering runtime constraints. The motivation behind development of CG is to (1) meet the criteria for acceptable performance standards in speed of execution, and (2) enable the application of GReAT system implementations as a feasible alternative to hand

written code, i.e. without the introduction of significant performance overhead. Clearly, the performance of the generated code will remain below that of the hand written code, but the reduced development time attributed to the implementation of the transformation using the GReAT approach often compensates or outweighs the conventional manual implementation techniques.

4.2 Partial Evaluation

If we write the Graph Rewriting Engine (GRE) of GReAT as a function it will have the following signature:

$$GRE : (I \times M_I \times M_O \times T) \rightarrow O \text{ , where}$$

- M_I, M_O - metamodels. A Metamodel is a graph that defines the graph grammar of the input/output models.
- *I* – input model. A graph that conforms to the metamodel M_I.
- *O*- output model. A graph that conforms to the metamodel M_O.
- T - transformation. Is a graph rewrite/transformation specification.

The Code Generator performs a partial evaluation of the GRE function to produce code specific to a given transformation and input/output metamodels.

$$CG : (M_I \times M_O \times T) \rightarrow (T_C : (I) \rightarrow O)$$

The justification for the partial evaluation is that the transformation and the metamodels make up the invariant part of transformation system. The same transformation is typically run on multiple inputs over a course of time. We argue that once the transformation and the modeling paradigm(s) reach a mature state, the transformation can be compiled into a high-performance executable that is capable of performing transformations in an efficient way.

By treating the metamodels as invariants, the CG can generate code that manipulate input and output models using paradigm-specific API's. These API's are generated by Universal Data Model (UDM), a framework that provides object-oriented C++ interfaces to programmatically access input/output models. UDM can generate a domain specific custom API with type-safe access methods for object creation/removal, link creation/removal, and attribute setters/getters [15]. The transformation executable can be built by compiling the generated transformation files and the paradigm-specific API files.

4.3 Implementation of Algorithmic Optimizations

The transformation rules are compiled into C++ class definitions. Although from the point of view of the language semantics, the procedural programming paradigm would suffice, we will see later in this section that introducing user-defined types for the transformation rules results in a much cleaner design. In C++, data abstraction is

implemented by classes, and the class is also the unit of encapsulation, which OO concept will assist in the implementation of the packet passing mechanism.

The only function exposed in the public interface of each class definition is the function operator; calling the function operator of a given class triggers the execution of the corresponding rule.

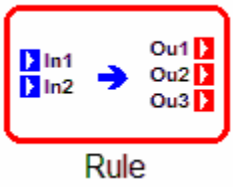
GReAT trans. Rule	Generated C++ class definition
	*class Rule* *{* *public:* *void operator()(const Packets_t& In1, const Packets_t& In2, Packets_t& Ou1, Packets_t& Ou2, Packets_t& Ou3);* … *};*

Fig. 5 Mapping GReAT transformation rules to C++ classes

GReAT introduces the optimization *"Reusing Previously Matched Objects"*, which is basically the idea of passing graph objects attached to *ports* from one rule to another.

The function operator argument list implements the facility for passing vertices of already matched subgraphs between a set of rules. *Packets_t* is a list of objects, which is the common base class for all classes that represent the graph objects. In this context, the *Packets_t* arguments represent a list of graph objects directed to a specific port. Subgraphs can be derived by taking together the input or output packets of the corresponding rule.

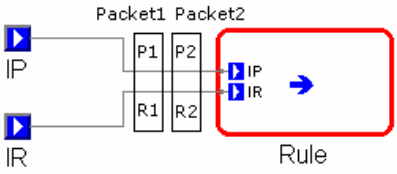
Two incoming packets, two input ports	Generated C++ interface
	*void operator()(* *// IP will contain P1 & P2* *const Packets_t& IP,* *// IR will contain R1 & R2* *const Packets_t& IR* *);*

Fig. 6 A graph object passing use-case

The implementation of *operator()* depends on the type of the GReAT rule:

- For rules of type *Block*, the function operator executes the contained rules in the sequencing order for all incoming packets.
- For rules of type *ForBlock*, the function operator executes each contained rule for each incoming packet, one-by-one.
- For rules of type *Test*, the contained cases are executed for all incoming packets in a deterministic order, which is derived from the physical placement of the cases, until a match has been found in a case.
- For rules of type *ForTest*, the contained cases are executed in the same way as in the case of the *Test*, but each case is executed for each incoming packet, one-by-one.

- For rules of type *Rule*, the incoming packets representing a host subgraph are tested against the pattern graph, then for each match new objects are created and matched objects deleted according to the rule specification. The pattern objects connected to the output port are then used to create output packets.
- For rules of type *Case*, incoming packets representing a host subgraph are tested against the pattern graph, and the pattern objects connected to the output port are then used to create output packets.

4.4 Rule Execution and Sequencing

The generated transformation code can be initiated starting from any rewriting rule; rules contained in that rule will be executed. The execution sequence of contained rules is maintained by *rule callers*. Rule callers are protected member functions of composite rules, and are designed to implement three important tasks:

1 Calling rules with the necessary arguments.
2 Calling the rule callers of *destination rules*. *Destination rules* are defined as the set of those rules, whose inputs are supplied by the current rule.
3 Forwarding packets to the output ports of the parent rule.

Let block *Block* contain *Rule0, Rule1, Rule2*. The class definition generated for block *Block*, will contain one rule caller for each rules: *Rule0, Rule1, Rule2*. Fig. 7 shows the code for the rule caller of *Rule0*. Observe, how packets are passed from caller to caller through the function argument lists.

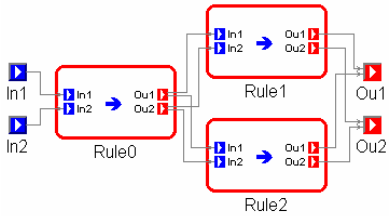
GReAT rule sequencing pattern	**Generated C++ code**
	*void Block::callRule0(const Packets_t& In1,* *const Packets_t& In2) {* *// declare output packets local* *Packets_t Ou1, Ou2;* *Rule0 rule0;* *rule0(In1, In2, Ou1, Ou2);* *if ((!Ou1.empty()) && (!Ou2.empty()))* *callRule1(Ou1, Ou2);* *if ((!Ou1.empty()) && (!Ou2.empty()))* *callRule2(Ou1, Ou2);* *}*

Fig. 7 Rule execution and sequencing via rule callers

To solve task 3, each contained rule must be able to append its own outputs to the parent block's outputs. The block's outputs should be visible only for those rules which are contained in that block. If every block output were passed as an individual argument to each rule caller function, the interface would quickly become very bloated. Our preferred approach to this problem is to represent the block output ports as member variables. Initially, rule callers had been implemented as procedures, but the design constraint described above led us to rely on OO encapsulation and C++ classes. Restricting data visibility to a set of functions automatically entails the

conversion of those functions to class member functions. Classes are the primary language elements in C++ that represent concepts in the application domain. Thus in the process of seeking a good representation of the GReAT rules, we eventually employed *abstract data types.*

4.5 Pattern Matcher

The graph rewriting/transformation process starts with the *pattern matching*, where an input subgraph and pattern graph are used to perform subgraph isomorphism on the input graph. The pattern graph can be described as a set of graph objects each with specific type, connected together with specific containment and association relationships. Therefore the task of the generated code is to (1) type-check all input subgraph elements and (2) check for the existence of the pattern vertices and edges in the input graph. If any type mismatch is found, then the pattern matching fails for the current input, and the processing proceeds to the next input. The pattern matching algorithm used in CG traverses the relationships specified in the pattern graph and generates code for each relationship. Traversed relationships are marked bound. The code generation process stops when all relationships in the pattern have been marked bound.

Specification of the types of the graph objects is fully exploited in the code generation process in the following way. In contrast with a naïve and general pattern matching implementation, we utilize strongly typed interfaces in the generated code, which leads to the restriction of the candidate objects and relations. This brings about the performance gain consequence of checking smaller number of graph objects and relations in the input host graph.

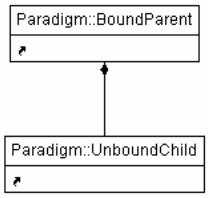
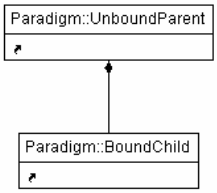
GReAT pattern	**Generated UDM API C++ pseudo-code**
	*//get children of type UnboundChild only* *set< Paradigm::UnboundChild> unboundChilds=* *boundParent.UnboundChild_kind_children();* *for(set< Paradigm::UnboundChild>::const_iterator it=* *unboundChilds.begin();it!= unboundChilds.end(); ++it){* *Paradigm::UnboundChild currUnboundChild = *it;* … *}*
	*Udm::Object& boundChildParent= boundChild.container();* *// UDM RTTI* *if(false== Uml::IsDerivedFrom(boundChildParent.type(),* *Paradigm::UnboundParent::meta))* *continue;* *Paradigm::UnboundParent unboundParent=* *Paradigm::UnboundParent::Cast(boundChildParent);* …

Fig. 8 Composite relationships and the respective generated code fragments

The GReAT definition of a match enforces that each pattern object must refer to a unique host graph object. The brute force approach would check a newly matched graph object with all previously matched objects before actually making the match. However, *identity checks* need not be so thorough, because in many cases different pattern objects cannot possibly refer to the same graph object. Objects of different types cannot be identical, objects of different parents cannot be either, and so on. Nevertheless there are cases, when they can, and they would, if *identity checks* did not prevent it.

The three fundamental relationships where identity checks are necessary:

1. Parent with two or more children, where two or more children are of the same type.
2. Simple associations such that a source has two or more destinations, where the destinations are of the same type; (source and destination roles are interchangeable)
3. An association class such that a source is connected to two or more association classes, where any association class has the type of another association class, or a source is connected to two or more different type association classes, but connected to destinations, where any destination has the type of another destination; (source and destination roles are interchangeable)

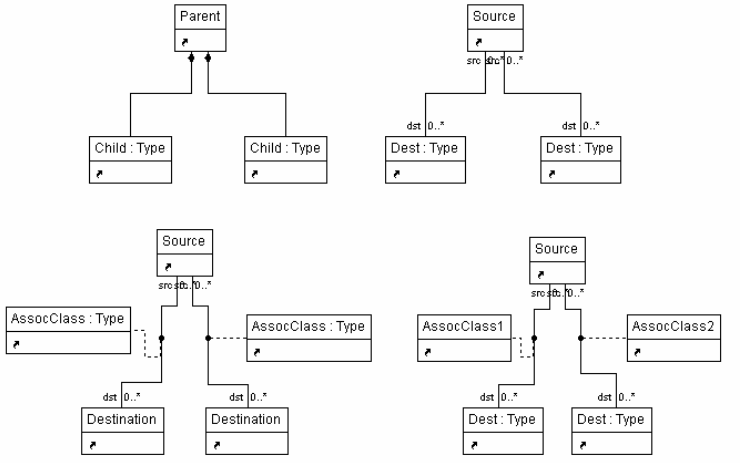


Fig. 9 Patterns where identity checker code is necessary

The performance gain resulting from the application of identity checkers is apparent for large pattern graphs along with those relationships described above. The brute force approach that checks each object for uniqueness would impose a significant performance overhead in the runtime.

The generated code is guaranteed to have unique objects in the match iff, the objects in the match have the same type, and they are connected to some identical object (such as a parent, or other end of an association). (‘Has the type’ means same type, or direct or indirect descendant of, as in OO terminology.)

4.6 Effector

Pattern graph can have pattern objects with roles set to *CreateNew* or *Delete*, as described in Section 3.1. The actions are executed for each match found by the pattern matcher. Fig. 10 presents some examples of consequence code generation.

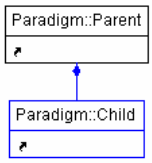
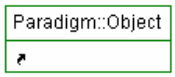
GReAT consequence	Generated UDM API C++ pseudo-code
	*// Create Pattern Object* *Paradigm::Child newChild= Paradigm::Child::Create(Parent);*
	*// Delete Pattern Object* *if (Object) // if Object exists* *Object.DeleteObject();*
	*// create multiple cardinality simple association link* *Source.dst()+= Destination;* *// create single cardinality simple association link* *Source.dst()= Destination;*

Fig. 10 GReAT consequences and the respective generated code fragments

Another specification element of consequences is the *Attribute Mapping Code*. These are code snippets provided by the user to manipulate the attributes of the graph objects. The specification language for these snippets is C/C++, hence the code can be directly copied into the generated code. The CG provides the context for the *Attribute Mapping Code* by instantiating variables with pattern object names within the scope of the *Attribute Mapping Code*.

4.7 Architecture of the CG

Having illustrated some required features of the generated code, we now focus our attention on the design aspects of the CG tool. We present a way of using the *composite* design pattern to produce the inherently complex transformation code. But before going into details we present the architectural overview of the CG tool.

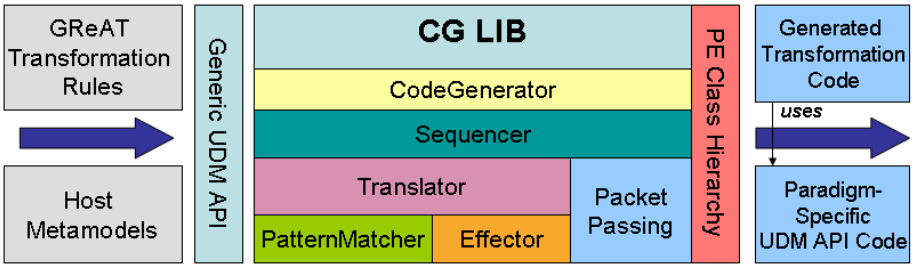


Fig. 11 The overall architecture diagram of the CG tool

The most fundamental design challenges along with their resolutions regarding the generated C++ code are:

1. *Simplicity and Clarity:* Introducing the OO programming paradigm, that results in a faithful representation of the architecture of the application domain. Dependencies between different parts of the program are minimized by the use of well-defined interfaces.
2. *Correctness and Safety:* The generated code is built up from small validated code blocks in an iterative manner. Type safety is ensured by performing run-time type checks previous to executing type casts. Application of STL containers and iterators, which are not only safe but also predictably efficient with the STL complexity guarantees.
3. *Efficiency:* Prudent application of language-level efficiency guidelines such as passing objects by references, omission of creation objects on the heap, elimination of implicit temporary objects, etc.

The approach we have taken to achieve these goals is to build a C++ syntax tree like structure from various abstract data types representing C++ language primitives, For instance, we created a class to embody the representation of a C++ class definition or a C++ for-loop. This family of classes can also be thought of as the metamodel for the output code, where each metamodel element corresponds to a specific code fragment. In the code generation process we essentially construct the appropriate objects and compose these objects into tree structures to represent the generated program hierarchy. Finally the resulting hierarchical structure (or part of) can be serialized in to C++ source file(s). This is an application of the *Composite* design pattern [16].

The key to this design is the class *PE*, which is declared abstract, and represents both primitives and containers. *PE* (Program Element) declares composite operations for managing its children, and a *print()* function, which prints the object to the stream, which is the function argument of *print()*.

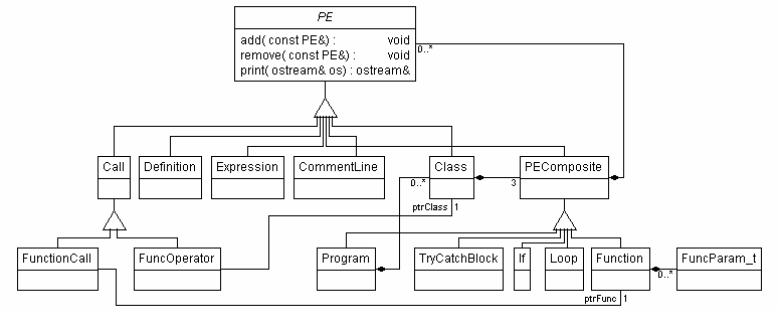


Fig. 12 The PE Class Hierarchy

All these operations are declared as virtual to take advantage of *polymorphism* in a variety of cases where type dependent run-time discrimination is needed. One example is the *print()* function of the composite class, which performs recursive serialization through the polymorphic container *_elems*.

```
std::ostream& PEComposite::print( std::ostream &os) const
{
        typedef std::list< const PE*> PEs_t;
        // print children
        for( PEs_t::const_iterator it= _elems.begin(); it != _elems.end(); ++it)
        {
                const PE* pe= *it;
                os= pe->print( os);
        }
        return os;
}
```

The CG tool generates human readable code -featuring formatted output and separate header/source serialization-, by simply invoking the *print()* method on the topmost object in the container hierarchy. Therefore, the fundamental challenge is rather the *creation* of the *PE* hierarchy. In this sense, the CG tool can be thought of as a translator, which translates the GReAT program hierarchy into the *PE* syntax hierarchy.

The various modules of the Code Generator perform this translation in different stages during the generation process. Each class encapsulates a set of algorithms specific to a given task. If one class uses some services of another class, we made that first class configurable with the behavior of the other class. This design first and foremost supports runtime configuration of the various components, but it also enhances reusability and advocates extensibility in the future. This is the *Strategy* design pattern [16]. For example, the Sequencer class can be configured with a Translator class, therefore the same sequencer can be (re)used with different translator implementations. The key components of the CG tool are described below:

- *CodeGenerator:* This class provides the main user interface for the code generation. It offers the family of strategy objects (see below) that support the default translation course. The class also contains a *Program* object, which represents the target of the translation. The primary design goal was to make this class as easy to use as possible; if the user wants to perform more sophisticated code generation, he can implement his own strategy classes and plug them in the existing code generation framework.
- *Sequencer:* The structure of the GReAT rules can be represented by a possibly cyclic directed graph. The Sequencer performs a Breadth-first traversal on this graph starting from the root rule. Already visited vertices are marked to prevent infinite traversals of the graph. For every sequenced rule, the Sequencer calls the Translator to generate code which implements that particular rule.
- *Translator:* This is the class that is primarily responsible for the translation. Each rule is translated to its equivalent C++ class definition. The translation procedure depends on the type of the rule, i.e. the implementation of the class is rule-type-dependent. It is the Translator that generates the rule callers, and it also keeps track of the mapping between the rules and the corresponding generated class definitions. The Translator uses PacketPassing, PatternMatcher and Effector modules to perform those subtasks that are not directly related to the generic translation process.

- *PacketPassing:* This module defines a set of data structures that keep track of the mapping between ports of a rule and the function argument list generated for that rule. PacketPassing plays an indispensable role in the generation of the rule callers, where function operators are to be called with the correct function argument list to support the passing of already matched subgraphs between various rules.
- *PatternMatcher:* The class PatternMatcher encapsulates all the translation logic which is associated with the pattern matching code generation. The pattern matching algorithm traverses the edges of the pattern graph, and produces optimized pattern matching program code which implements the pattern specification. The generated code checks the types of the graph objects and the existence of specific type relationships in the host subgraph. in an efficient way which is described in Section 4.5 The input subgraph gets entered into the pattern matching context through the use of the PacketPassing data structures. The PatternMatcher also generates an embedded class definition which represents the match specific for the given rule. If a match has been found, a container of the matches is appended with the match. This container is going to be used in the implementation of consequences, which is generated by the Effector.
- *Effector:* Consequences, such as object creation, deletion, and creation or deletion of relationships are implemented by the Effector class. This class also responsible for printing out the Attribute mapping code and creating output packets from those pattern objects which are connected to the output ports of the related rule.

The generated code compiles without any custom modifications, and it is also ISO Standard C++ compliant, consequently platform independent.

4.8 Related work

The PROGRES environment developed at Aachen University of Technology solves the pattern matching efficiency problem with the help of a simple heuristic optimization algorithm, which is based on the implementation of a sophisticated cost model. PROGRES, similarly to GReAT, introduces various language elements to restrict the number of possible search paths, like node and edge types and edge cardinality assertions [18]. PROGRES also offers an interpreter and a cross-compiler, the later of which generates efficient Modula-2 or C code for PROGRES specifications [7].

OPTIMIX is a flexible optimizer generator developed by Uwe Aßman at University of Karsruhe. Its input language is based on Datalog and graph rewriting systems such as edge addition rewrite systems (EARS) and stratified graph rewrite systems (stratified GRS). The transformation can manage data models specified in heterogeneous syntax formats like Java, AST and CoSy-fSDL. OPTIMIX is capable of generating C or JAVA code based on the type of the input data models [17].

CLEAN, created by the Software Technology department of the University of Nijmegen, is a functional programming language with explicit graph rewriting semantics. A CLEAN program basically consists of a number of graph rewrite rules

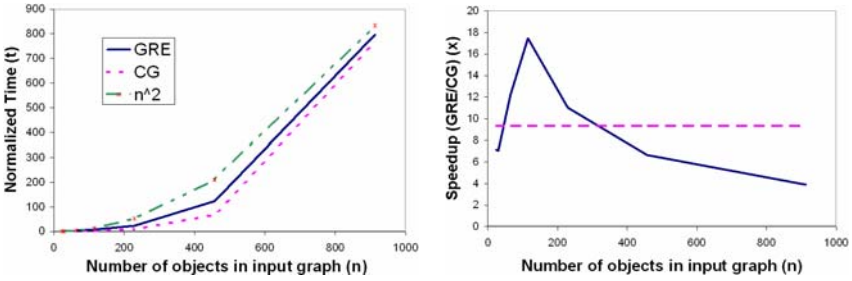
with the common graph transformation semantics, but the left-hand side graph being required to be a tree. CLEAN is not primarily designed to be a sophisticated graph transformation specification language. Rather it is a modular and general purpose programming language offering I/O libraries and diverse type systems [19].

Whereas GReAT, PROGRES and OPTIMIX generate code in other programming languages and use external general-language compilers to generate executables, CLEAN provides its own compiler and code generator, producing efficient native object code optimized for graph rewriting programs.

5 Comparison of CG with GRE

In this section we will present some a comparison of the execution time of the GRE and the Code Generator. Two transformation problems have been chosen for the comparison. These transformations are:

1. *Df→Fdf:* Transform Hierarchical dataflow to its equivalent Flat dataflow representation.
2. *Hsm→Fsm:* Transform Hierarchical Concurrent State Machine (HCSM) to its equivalent Finite State Machine (FSM).

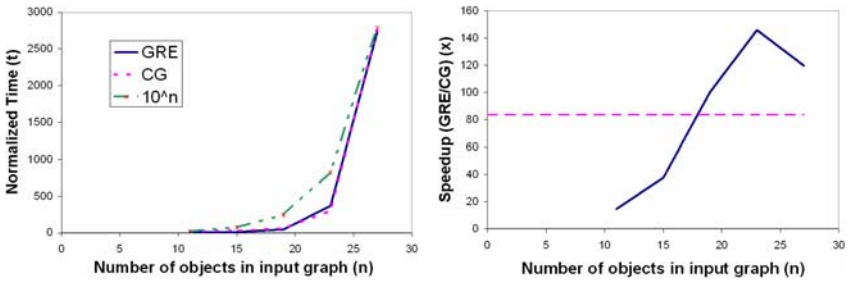


(a) Normalized performance **(b) Code Generator speedup**

Fig. 13 Performance graphs for Df→Fdf

To evaluate the performance of CG in comparison with GRE, the Df→Fdf transformation was executed on 7 different input graphs. The size of these graphs varied from 24 vertices to 914 vertices. Execution times of GRE and CG were measured for all the inputs. Fig. 13(a) is a plot of the input graph size (n) vs. normalized execution time for both GRE and CG. Matlab's polyfit function was used to find the closest fitting polynomial or exponential to the results and the second order polynomial yielded the best results. For this reason the n^2 plot is also shown in Fig. 13. From the graph we can see that the order complexity of the transformation doesn't change significantly between GRE and CG and is governed by the complexity of the transformation algorithm. Experimentally we have seen that the transformation algorithms complexity is $\sim O(n^2)$. Fig. 13(b) shows the graph of n vs. speedup achieved by the code generator. The dashed line in the graph represents the average

speedup of 9.3x. From the graph we can see that the speedup varies within a bound ranging from 4x to 18x.



(a) Normalized performance **(b) Code Generator speedup**

Fig. 14 Performance graphs for *Hsm→Fsm*

For *Hsm→Fsm*, 4 input graphs were used. These graphs only had parallel states and varied from 11 vertices to 27 vertices. Execution times of GRE and CG were measured for all the inputs. Fig. 14(a) is a plot of the input graph size (n) vs. normalized execution time for both GRE and CG. The polyfit function was again used and this time an exponential to the base 10 yielded the closest results. For this reason Fig. 14 also shows the 10^n plot. From the graph we can see that the order complexity of the transformation doesn't change between GRE and CG and is governed by the complexity of the transformation algorithm. In this case we see that that transformation algorithms complexity approaches $\sim O(10^n)$. Fig. 14(b) shows the graph of n vs. speedup achieved by the code generator. The dashed line in the graph represents the average speedup of 83.3x. From the graph we can see that the speedup varies within a bound ranging from 14x to 119x. The 14x speed up was observed for very small models and could be because of a constant runtime overhead. A speedup of ~100x was observed consistently for larger models.

From the experiments we see that the user is able to specify transformations with polynomial characteristics, this can be attributed to the language features provided in GReAT. On the other hand exponential algorithms can also be specified as in the case of *Hsm→Fsm.*

The second conclusion is that the order complexity of the transformation remains the same for both GRE and CG. This is an expected result because the code generator doesn't not perform any modifications that can provide a gain in order complexity.

The speedup doesn't seem to have a definitive trend with respect to the input size but vary a lot from one kind of transformation to another. Df→Fdf, an $O(n^2)$ transformation yielded an average speedup of ~9x while the Hsm→Fsm, an $O(10^n)$ transformation yielded an average speedup of ~85x. These results make us believe that the speedup is dependent on the complexity of the transformation. For higher complexity transformations the speedup is also higher.

One possible reason for such a result can be based on the percentage of the total execution time spent in the pattern matching as opposed to the packet passing and other housekeeping work. Since a higher order complexity algorithm will spend more

time in the pattern matcher, and the code generator partially evaluates the pattern matcher, a better speedup is observed. When the time complexity of the algorithm is small and the size of the models is large. The packet-passing/housekeeping overhead is a large percentage of the total execution time and the speedup observed is less.

6 Conclusion and future work

Specification of Graph transformations using high-level transformation languages has many advantages in Model Driven Architecture (MDA), tool integration and other areas in software engineering. The major bottleneck associated with graph rewriting systems is poor runtime performance. Performance issues need to be tackled at all levels of the transformation system, ranging from the language to low-level implementations. There are two major categories of optimizations: (1) language and algorithmic, that may yield an improvement in the order complexity and (2) partial evaluation and implementation optimization that produce a constant time improvement.

Three language-level optimizations have been described in the paper – (a) typed patterns, (b) Pivoted pattern matching and (c) Reusing previously found objects. These optimizations on an average can produce a significant speedup in the execution time of the transformations. As shown in the comparison section, the Df→Fdf transformation had an order complexity of only n^2 for both GRE and CG.

Although the reduced development time attributable to the graph specific language semantics is an obvious benefit, efficiency drawback in the execution time can still prevent the application of graph transformations in many real systems. We have described how partial evaluation and other implementation techniques can considerably speed up the transformations. In the case of Hsm→Fsm transformation the CG provided a speedup of ~100x over the generic graph rewrite engine.

Though many solutions have been presented in this paper to address the performance needs, there are still some transformations that are exponential. In these case though we cannot change the order complexity of the transformation we should be able to further optimize the implementations such that they produce much better results in the average case. Until the generated code does not do better than its hand coded counterpart in speed of execution, there will be always room for improvement. The generator can factor out repetitive transformations and reuse parts of the pattern matching code.

7 Acknowledgement

The DARPA/IXO MOBIES program, Air Force Research Laboratory under agreement number F30602-00-1-0580 and NSF ITR on "Foundations of Hybrid and Embedded Software Systems" programs have supported, in part, the activities described in this paper.

8 References

[1] J. Sztipanovits, and G. Karsai, “Model-Integrated Computing”, Computer, Apr. 1997, pp. 110-112.
[2] J. Rumbaugh, I. Jacobson, and G. Booch, “The Unified Modeling Language Reference Manual”, Addison-Wesley, 1998.
[3] “The Model Driven Architecture”, http://www.omg.org/mda/, OMG, Needham, MA, 2002.
[4] “Request For Proposal: MOF 2.0 Query/Views/Transformations”, OMG Document: ad/2002-04-10, 2002, OMG, Needham, MA.
[5] Grzegorz Rozenberg, “Handbook of Graph Grammars and Computing by Graph Transformation”, World Scientific Publishing Co. Pte. Ltd., 1997.
[6] Blostein D., Schürr A., “Computing with Graphs and Graph Rewriting”, Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.
[7] A. Schürr, “PROGRES for Beginners”, Technical Report, Lehrstuhl für Informatik III, RWTH Aachen, Germany.
[8] H. Gottler, “Attributed graph grammars for graphics”, H. Ehrig, M. Nagl, and G. Rosenberg, editors, Graph Grammars and their Application lo Computer Science, LNCS 153, pages 130-142, Springer-Verlag, 1982.
[9] J. Loyall and S. Kaplan, "Visual Concurrent Programming with Delta-Grammars," Journal of Visual Languages and Computing, Vol 3, 1992, pp. 107-133.
[10] G. Engels, H. Ehrig, G. Rozenberg (eds.), “Special Issue on Graph Transformation Systems”, Fundamenta Informaticae, Vol. 26, No. 3/4 (1996), No. 1/2, IOS Press (1995).
[11] H.Ehrig, M. Pfender, H. J. Schneider, “Graph-grammars: an algebraic approach”, Proceedings IEEE Conference on Automata and Switching Theory, pages 167-180 (1973).
[12] Agrawal A., Karsai G., Shi F., “A UML-based Graph Transformation Approach for Implementing Domain-Specific Model Transformations”, Technical report, (ISIS), Vanderbilt University, Nashville, TN, 2003.
[13] Agrawal A., Karsai G., Ledeczi A., “An End-to-End Domain-Driven Software Development Framework”, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 26, 2003.
[14] Karsai G., Agrawal A., Shi F., Sprinkle J., “On the Use of Graph Transformations for the Formal Specification of Model Interpreters”, JUCS, November 2003.
[15] Magyari E., Bakay A., Lang A., Paka T., Vizhanyo A., Agrawal A., Karsai G., “UDM: An Infrastructure for Implementing Domain-Specific Modeling Languages”, The 3rd OOPSLA Workshop on Domain-Specific Modeling, OOPSLA 2003, Anahiem, California, October 26, 2003.
[16] Gamma, R. Helm, R. Johnson and J. Vlissides, “Design Patterns”, Addison-Wesley, 1995.
[17] Aue Aßmann, “OPTIMIX, A Tool for Rewriting and Optimizing Programs”, Technical Report, University of Karslruhe, Germany, 1998.
[18] Albert Zundorf: “Graph Pattern Matching in PROGRES”, Graph Grammars and Their Application to Computer Science, 5h International Workshop, Williamsburg, VA, USA 1994.
[19] “CLEAN: Version 2.0 Language Report”, Software Technology department, University of Nijmegen, The Netherlands,