Reusable Idioms and Patterns in Graph Transformation Languages

Aditya Agrawal, Attila Vizhanyo, Zsolt Kalmar, Feng Shi, Anantha Narayanan, and Gabor Karsai

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA
{Aditya.Agrawal,Zsolt.Kalmar,Feng.Shi, Anantha.Narayanan,Attila.Vizhanyo,Gabor.Karsai}@vanderbilt.edu

Abstract. Software engineering tools based on Graph Transformation techniques are becoming available, but their practical applicability is somewhat reduced by the lack of idioms and design patterns. Idioms and design patterns provide prototypical solutions for recurring design problems in software engineering, but their use can be easily extended into the graph transformation systems. In this paper we briefly present a simple graph transformations language: GREAT, and show how typical design problems that arise in the context of model transformations can be solved using its constructs. These solutions are similar to software design patterns, and intend to serve as the starting point for a more complete collection.

1 Introduction

The practical application of Graph Rewriting and Transformations (GRT) [4] is contingent upon the existence of mathematically well-founded, yet easy-to-use tools on one hand, and on the real-world engineering experience and knowledge about the use of the techniques on the other hand. With the arrival of the Model-Driven Architecture (MDA) [3], GRT is about to become a technology that could be widely used in the industry. Although there have been a number of GRT tools developed [18, 19], few tools have been used on practical development projects, and even lesser engineering experience has been accumulated and documented about the use of these tools.

We agree with the vision of MDA, where transformations on the artifacts produced during the design of software are an integral and essential part of the design process. We envision software development environments, where model transformations are used to facilitate *design automation.* Transformations could take place at different phases of the process, for instance: (1) when design models are built and some activities (e.g. applying a design pattern) are best implemented by an automated tool that transforms the models [16],(2)when components are adapted to suit the needs of a particular design context [20], (3) when designs have to be transformed into a model that can be analyzed by an analysis tool [10], (4) when code has to be generated from the models through instantiating code fragments [21]. As these transformations must be performed on design models (which are typed multi-graphs in the most general sense), GRT techniques are applicable.

There are (at least) two major motivations for using GRT in this context: (1) transformations could be complex, and a concise and precise language to program them is desirable, and (2) if the transformations are specified in a formal way (as the GRT technology allows it), we have the opportunity to reason about their properties, and how they change the properties of the models they are applied to.

A language to write graph transformation programs in (and thus implement model transformations), should have a well-defined, yet simple syntax and semantics. However, there are common recurring tasks in model transformations that should not be directly supported by the language. Rather, they should be available as well-documented, reusable idioms and design patterns that solve recurring design problems. The difference between the two is that idioms are restricted to an application domain, while design patterns are domain-independent. In this paper we describe a few such design patterns and idioms. First we describe a simple visual language that supports explicitly sequenced graph transformation and rewriting operations. Next a number of domain-independent design patterns are described, followed by a description of a non-trivial idiom. The final sections discuss related and future work.

2 GReAT

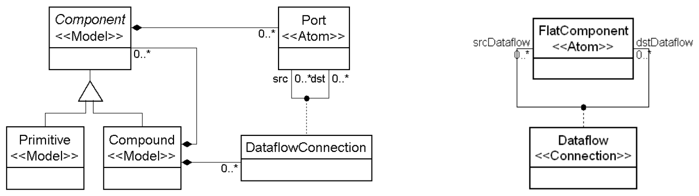
The transformation language used to demonstrate the design patterns, algorithms and idioms is Graph Rewriting and Transformation language (GReAT). GReAT is a language that allows users to specify graph transformations in a graphical form with formal and executable semantics. In this paper only the necessary language constructs are explained, [7] describes the full approach and support tools. The operational semantics of GReAT is formally defined in [9]. GReAT is based on the theoretical work of graph grammars and transformations [4–6] and belongs to the set of practical graph transformations systems, like AGG and PROGRES. This language can be divided into four parts:

1. Domain specification and heterogeneous transformations
2. Pattern specification language
3. Graph transformation language
4. Control flow language.

2.1 Domain Specification and Heterogeneous Transformations

Many approaches have been introduced in the literature to capture graph domains. For instance, schemas are used in PROGRES while AGG uses type graphs. These are proprietary formats for the specification of the graph domain.

We chose UML [1] class diagrams and the Object Constraint Language (OCL) [2] for the specification of domains because it is standardized and it is more expressive than both schema and type graphs. The UML class diagram plays the



(a) Hierarchical dataflow (b) Flat dataflow

Fig. 1. Metamodels HDF and FDF

role of a graph grammar such that it can describe all the “legal” object networks that can be constructed within the domain.

From the UML class diagrams one can generate an object oriented API that can be used to implement the graphs, to traverse the input graph, and to construct the output graph. To satisfy the second requirement, GReAT allows the user to specify any number of domains that can be used for the transformation purposes.

For example, figure 1(a) shows a UML class diagram that represents the domain of hierarchical data flow networks. A hierarchical data flow network consists of *Components*, *DataflowConnections* and *Ports*. A component can be a either a *Compound* or a *Primitive* component, a *Compound* component may contain other components. Components contain port and directed dataflow associations between these ports represent the flow of data. figure 1(b), represents a flat (non-hierarchical) dataflow language with FlatComponents and dataflow connections between them. The hierarchical data flow network and flat representation will serves us as an ongoing example throughout this paper.

A design challenge for GReAT was to provide a uniform syntax and semantics for both graph transformations and rewriting. This problem is tackled in GReAT by allowing the user to compose source and target metamodels by defining temporary vertex and edge types that can span across multiple domains and will be used temporarily during the transformation. For example, figure 2 shows a metamodel that defines associations/edges between HDF and FDF. Component and Dataflow are classes from figure 1(a) while the FlatComponent and FlatDataflow are classes from figure 1(b). This metamodel defines three types of edges. There is a refersTo edge type that can exist between Component and FlatComponent, and between Dataflow and FlatDataflow. Another edge type associatedWith is defined and it can link Component objects. By composing the domains using temporary cross-links we are able to tie the different domains to-

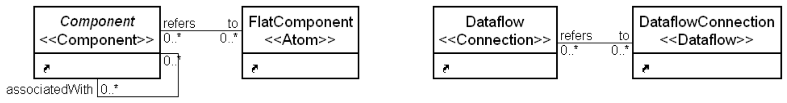


Fig. 2. A meta-model that introduces cross-links

gether to make a larger, heterogeneous domain that encompasses all the domains and cross-references.

2.2 The Pattern Specification Language

The pattern specifications found in graph grammars and transformation languages [4–6] were not sufficient for our purposes. A more expressive, easy-to-use pattern language has been developed that allows specification of complex graph patterns. The pattern specification language was developed to extend simple patterns with a notion of cardinality on each pattern vertex and each edge. Precise semantics for such a language was developed along with efficient pattern matching algorithms. For a complete discussion on semantics, expressiveness and matching algorithms of pattern graphs please see [7].

2.3 Graph Transformation Language

The heart of GReAT is the graph transformation language. It was inspired by many previous efforts such as [4–6]. It defines the basic transformation entity: a production/rule. A production contains a pattern graph, in which each pattern object: a vertex (or an edge) conforms to a type: a class (or an association) from the metamodel. Apart from this, each pattern object has another attribute that specifies the role it plays in the transformation. There are three roles that a pattern object can play:

1. *bind*: The object is used only to match objects in the graph.
2. *delete*: The object is used to match objects, but once the match is computed, the objects are deleted.
3. *new*: New objects are created after the match is computed.

The execution of a rule involves matching every pattern object marked either *bind* or *delete*. If the pattern matcher is successful in finding matches for the pattern, then for each match the pattern objects marked *delete* are deleted and then the objects marked *new* are created.

Pre-conditions are often required for additional constraints on the transformation application. In GReAT, OCL is used for the pre-condition specification and these constraints are evaluated on the matches before the actions are applied. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing objects. “Attribute mapping” is a specification of such attribute manipulation and is executed after the transformation

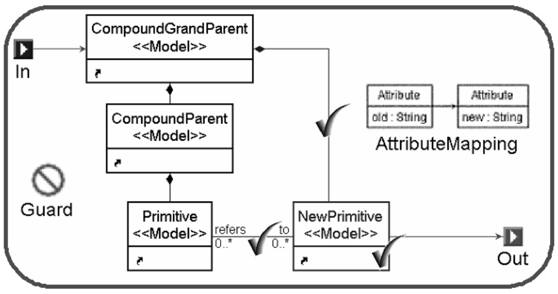


Fig. 3. Example rule with patterns, guards and attribute mapping

is applied. Figure 3 shows an example rule. Each object in the pattern graph refers to a class in the heterogeneous metamodel. The semantic meaning of this reference is that the pattern object should match with a graph object that is an instance of the class represented by the metamodel entity. The default action of the pattern objects is bind. The new action is denoted by a tick mark on the pattern vertex (see the vertex NewPrimitive in figure 3). Delete is represented using a cross mark (not shown in figure). The In and Out icons in the figure are used for passing graph objects between rules and will be discussed in detail in the next section.

2.4 Controlled Graph Rewriting and Transformation

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. In this paper these issues will be briefly touched upon, for a complete discussion please refer to [8].

The performance of the pattern matching can be significantly increased if some of the pattern variables are bound to elements of the host graph before the matching algorithm is started (effectively providing a context for the search). The initial matches, called pivots, are provided to a transformation rule with the help of ports that form the input and output interface for each transformation step. Thus a transformation rule is similar to a function, which is applied to the set of bindings received through the input ports and results in a set of bindings over the output ports. For a transformation to be executed graph objects must be supplied to each port in the input interface. In figure 3 the In and Out icons are input and output ports respectively. Input ports provide the initial match to the pattern matcher while output ports are used to extract graph objects from the rule so that they can be passed along to the next rule. The rules thus operate on packets, which are defined as sets of (port, host graph object) pairs.

Explicit sequencing of rules and a high-level control flow language allows the precise control of transformations and thus helps to manage the complexity of the transformation and allows users to write efficient transformations. The control flow language supports the following features:

1. Sequencing: rules can be sequenced to fire one after another.
2. Non-Determinism: Non-deterministic parallel execution of rules.
3. Hierarchy: Compound rules can contain other rules.
4. Rule reuse: The same rule can be called from different parts of the transformation specification.
5. Recursion: A level rule can (directly or indirectly) call itself.
6. Test/Case: A branching construct to choose between control flow paths.

Sequencing is used to specify an order of execution for a set of transformation rules. For example, figure 5 shows a sequence of rules, "HasComponents" and "Call: CollectPrimitives" are executed sequentially which is in parallel with the rule "IsPrimitive". Hierarchy is also shown in figure 5, where the above mentioned rules are all contained in a compound rule called the "CollectPrimitives".

A test/case construct is used to choose between different execution paths, similar to the 'if' statement in programming languages. Figure 11, contains a test called "TestProxyExistence" that contains two cases (shown in figure 12). The test will first try case "HasProxy", if "HasProxy" succeeds then the outputs will be passed to the respective output ports and similarly for "NoProxy". Once all inputs have been evaluated the next rules in the sequence will be executed.

3 Patterns and Idioms for Reusable Graph Transformations

Software design is commonly regarded as the most difficult stage in the software development cycle. The goal is to design a system such that it is flexible, robust and reusable. Some design challenges are common and have been faced by many software designers. Over the years elegant solutions to such problems have been identified and implemented. When such a design solution is formally documented, identifying the participating elements, their roles and collaborations, and the distribution of responsibilities, a design pattern arises. A design pattern allows some aspect of system structure vary independently of other aspects, thereby promoting robustness and domain-independent reuse [13].

The same driving forces exist in the area of programming graph transformations and in the following section a few transformation patterns, algorithms and idioms will be introduced. Each pattern/idiom will be described in a uniform way with the following structure: (1) Motivation: a problem, where the need for the pattern arises. (2) Applicability: the general class of problems where the pattern is applicable. (3) Structure: the abstract specification of the pattern. (4) Benefits: the advantages of applying the pattern. (5) Known uses: a set of transformations where the pattern has been known to be applied.

A single motivating example will be carried throughout the design pattern discussion, and that is the flattening of hierarchical dataflow (HDF) to a flat dataflow (FDF) representation. In figure 1(a) the meta-model of the HDF has been presented with primitives that capture dataflow behavior, and compounds that are only used for encapsulating other components. The aim is to convert the tree structure of HDF to an FDF representation (see figure 1(b)) while preserving the dataflow connectivity. A simple algorithm is: (1) collect all primitive nodes and copy them to FDF, (2) trace the dataflow connection from each port in each primitive to a corresponding target primitive port, (3) replace this trace by a single dataflow association in FDF.

3.1 The Leaf Collector Pattern

Motivation. In step 1 of the HDF flattening algorithm, a requirement is to collect all leaf nodes in the hierarchy. For example, in figure 4, given the root component C we need to find all leaf primitives P_{121}, P_{11}, P_{21}, P_{22}. Figure 5

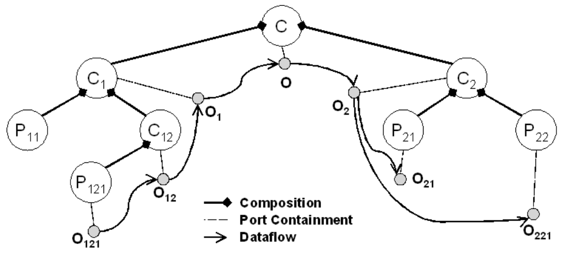


Fig. 4. An HDF model

shows the rules that collect all the primitives in a given HDF hierarchy. The top level rule "CollectPrimitives", gets as an input the root object of the hierarchy. It calls "HasComponents" that collects all direct children and on each child a recursive call to "CollectPrimitives" is made. If the input to "CollectPrimitives" is a primitive, then the "IsPrimitive" rule will succeed and will be passed to the output. At the end of the recursion, all primitives will be available at the output of the top level call.

Applicability. From a starting object, traversal of a particular kind of directed association is required till leaf objects are reached. Leaf objects are defined as objects, from which the association cannot be traversed any more.

Structure. The participants of this pattern are shown in figure 6. The "GetDirectNeighbors" rule is responsible for collecting all the direct neighbors of the input object. The "IsLeaf" rule is responsible for identifying if the input object is a leaf. This is achieved by a pattern with pattern cardinality equal to zero (see arrow in "IsLeaf"). The zero pattern cardinality on an association means that the two objects should not contain the specified association, and it is also known as the negative application condition. This implementation of "IsLeaf" is more general than the one seen in the dataflow example.

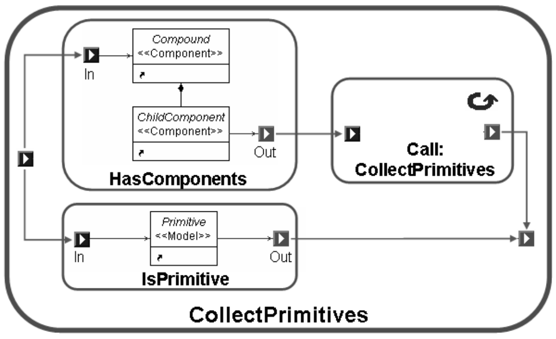


Fig. 5. Collecting primitives in HDF

Benefits. The traversal scheme and the leaf recognition are independent of each other. The processing of the leaves is separate from collecting the leaves and is typically done in a rule following the leaf collector pattern, and thus leaf collection and leaf processing can vary independently.

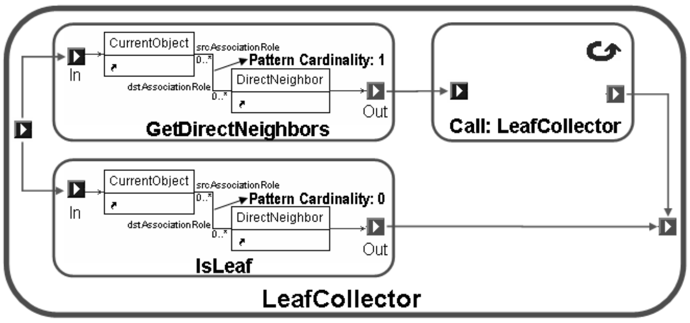
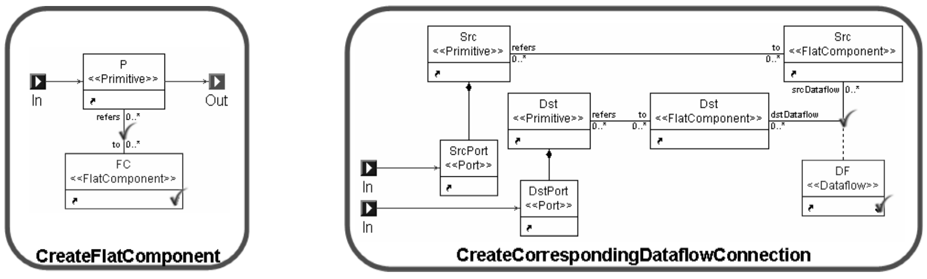


Fig. 6. The structure of the Leaf Collector pattern

Known Uses. (1) Collecting primitives in hierarchical structures. (2) Starting from a port finding all ports at the end of the dataflow connection chain. For example, in figure 4, given port O_{121}, find O_{21} and O_{221}.

3.2 The Map-Using-Link Pattern

Motivation. In flattening HDF, step 1, given a set of primitives, we need to create components in FDF. In step 2, given the source and destination ports of a dataflow connection in HDF, we need to find the corresponding source and destination ports in FDF, and make the equivalent dataflow connection. In traditional programming a map would have been used to store the correspondence between HDF and FDF objects. In a graph transformation language graphs are the only data structure. Thus the maps should be encoded as graphs. In figure



(a) (b)

Fig. 7. Rules copying dataflow connections

7(a) the rule "CreateFlatComponent" creates the corresponding FDF component for a given HDF primitive. The rule also creates a temporary association between *P* and *FC*. Creation of this association is functionally equivalent to the addition of an entry into a map. In figure 7(b) the rule "CreateCorresponding-DataflowConnection" creates a corresponding association in FDF for a given pair of source destination ports. This is achieved by traversing the temporary associations from the parents of these ports, which is functionally equivalent to the map lookup operation.

Applicability. When it is necessary to store/lookup correspondence between objects from different paradigms in the course of the transformation.

Structure. A temporary association type needs to be defined between the two types of objects in question. Then, in the rule where the corresponding object is created/identified the temporary association is created. In subsequent rules, to lookup the corresponding object, the temporary association is used to find the correspondence.

Benefits. Temporary associations that are used for representing the correspondence between objects are specified in a different UML class diagram. The implementation is such that the temporary associations are managed separately, leaving the source and target diagrams intact. Using graphs to represent these temporary data structures provides the flexibility of implementing any data structure required, without loss of generality. Since the temporary data structures are treated in the same manner as the source and the target graphs, the transformation language remains simple.

Known Uses. In every transformation where there is more than one graph being manipulated, and correspondence between the graph objects needs to be maintained.

3.3 Transitive Closure

Motivation. In the FDF a data dependency analysis needs to be performed. For such an analysis the transitive closure [23] (see figure 8) of the dataflow connection needs to be found. Figure 9 shows the transformation to compute the

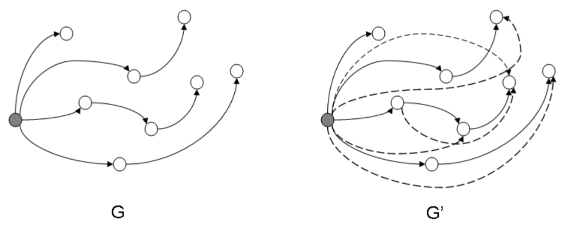


Fig. 8. A directed graph G and its transitive closure G'

transitive closure on FDF. The first step of the algorithm is to find all FDF components that do not have any incoming associations. This is achieved in "FindSource" using the zero cardinality association pattern. These FDF components are used as the initial value for the "TransitiveClosureStep". In one "TransitiveClosureStep", given a set of components, all their next and previous neighbors are found. If there is no association from the previous neighbor to the next neighbor, a new association is created between them. The "TransitiveClosureStep" is called again with the set of next neighbors as input.

Applicability. In all situations where the transitive closure needs to be computed.

Structure. The algorithm presented in figure 9 works only on directed acyclic graphs. Preprocessing steps can be used to convert an arbitrary graph into a directed acyclic graph by reducing all strongly connected components to a single vertex. In the general case the type of the association and the type of the vertex can be changed to suit the requirements of the problem.

Known Uses. Used to perform reachability analysis of distributed and parallel systems and in the construction of a parsing automata.

3.4 The Proxy Generator Idiom

So far this paper has discussed design patterns that have applicability to a large set of problem domains. This section will discuss the proxy generator design idiom, which is also reusable, but is restricted to a particular problem domain.

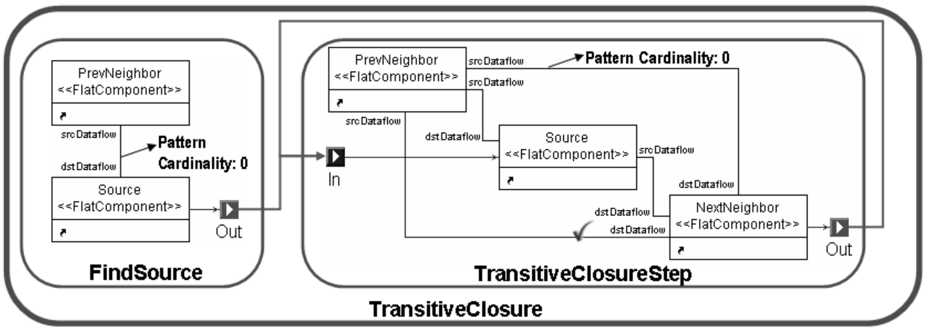


Fig. 9. Transformation rules to find the transitive closure

Motivation. A model for a distributed service based system (DSBS) is shown in figure 10. The distributed system (*System*) is composed of multiple processors (*Processor*), each processor hosts different objects (*Object*), and objects may request service (*Request*) from other objects. An object could be a proxy (*Proxy*) of a remote master object (*Master*) on a local processor, and such a relationship could be represented by the association *Distribute* between two objects. Therefore a proxy is a placeholder for another object and provides access control to it. In such distributed systems, in order to reduce network traffic as well as to

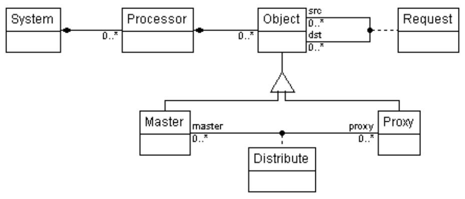


Fig. 10. UML class diagram of a general distributed system

abstract network communication from object interactions, we can use proxies on each processor to represent remote components locally. This optimization can be performed by identifying the need for proxies using static analysis methods on the object network, and then automatically generating them.

Figure 11 shows the transformation that identifies the need for proxies and creates them. For each *Master*, *Client* pair, such that the *Client* needs to make requests on the remote *Master*, the first step is to determine whether the *Master* has a proxy on the *ClientProcessor*. This is done in "TestProxyExistence". If it succeeds in finding a proxy, then "AssociateWithProxy" is called, which replaces the request to the master with a request to the local proxy. Otherwise, "CreateProxy" is called, which creates the equivalent proxy on the client's processor, followed by a call to "AssociateWithProxy".

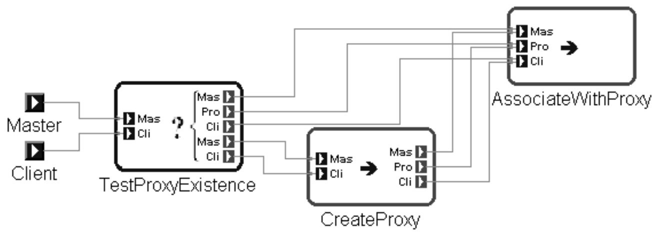


Fig. 11. Transformation rules for the proxy generator idiom

Step 1. Determine proxy presence on local processor: This is achieved using a test called "TestProxyExistence". It has two cases, "HasProxy" which checks the existence of a proxy for the *Master* in the *ClientProcessor*. If "HasProxy" is successful, the *Master*, *Client*, and *Proxy* are passed to the "AssociateWith-Proxy" rule. Otherwise the "NoProxy" case is called, that always succeeds and will pass the *Master*, *Client* pair to the next rule.

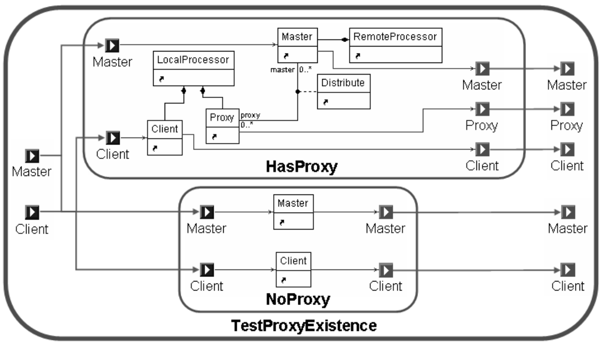


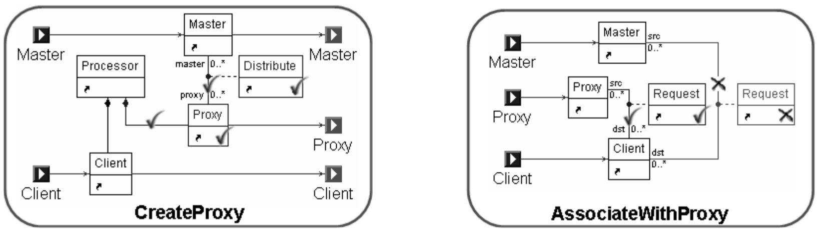
Fig. 12. TestProxyExistence

Step 2. Create the proxy (optional): In the rule "CreateProxy" as shown in figure 13(a), given the *Master* and *Client*, a *Proxy* is created on the client's local processor along with a distribute association with the *Master*. The newly created *Proxy*, its corresponding *Master* and the client are passed to the "AssociateWithProxy" rule.

Step 3. Migrate the services: In the rule "AssociateWithProxy" as shown in figure 13(b), given the Master and the corresponding Proxy, the clients request to the master is replaced by the clients request to the proxy.

Applicability. In any distributed system where remote interactions need to be abstracted and optimized. These interactions could be service/request, event source/sink, and dataflow interactions. The idiom can be used to perform a static analysis on a network of distributed objects for optimization, or can be used at runtime when the need for remote interaction arises.

Structure. In the general case the idiom has three distinct parts. (1) Given a master-client pair, test for the presence of the proxy. (2) If proxy is present, remove client-master association and create client-proxy association. (3) If proxy not present, first create a proxy and link it with the master, then remove client-master association and create the client-proxy association.



(a) CreateProxy rule (b) AssociateWithProxy rule

Fig. 13. Create and associate proxy

Consequences. The pattern separates the proxy identification, proxy creation and proxy association steps such that each can be modified independently.

Known Uses. Used in the Embedded System Modeling Language (ESML) [22].

4 Related Work

Software design patterns were documented since the early 1990's. The pioneering work was [12] where C++ idioms were introduced. However, it was [13] that contributed best of all to the acceptance and extensive use of patterns in object oriented design. In [14] 'generic patterns' or 'pattern templates' demonstrated extensible designs in generic languages.

Graph transformation has been used to automate the application of design patterns. [16] describes an approach which uses graph queries and graph rewriting rules to apply design patterns. In particular, a PROGRES-based tool is presented that transforms an application such that it conforms to a specific pattern. In [17] the suitability of GRS for the specification and execution of complex transformations on the static aspects of design is argued. Examples in the paper illustrate how GRS can be used to implement the design transformations proposed by MDA. [15] proposes to allow particular portions of the design to be specified in an abstract manner, with the goal of using these as patterns in order to search for corresponding implementation components. GRS is introduced as a formal method to perform the search and integration of those components.

The adoption of design patterns to Graph Rewriting Systems (GRS) has just started recently. Hence little work exists in this area.

5 Conclusions

We have shown how the idea of design patterns applies in the context of graph transformation programs. We demonstrated the concept using three (domain-independent) design patterns and one (domain-dependent) idiom. We believe that documenting design patterns in this manner is useful for practitioners of graph transformations and we call the community to contribute to this body of knowledge. Further research in this area could lead to comprehensive compilation of design patterns and idioms that could be used to educate developers using the graph transformation technology. At this time, our tool GReAT does not have direct support for design patterns. However, we believe that templatized rules (similar to template classes in C++) could serve as the foundation for design patterns. However, the implementation of this capability is a subject for future research.

6 Acknowledgement

The DARPA/IXO MOBIES program (F30602-00-1-0580) and the NSF ITR: "Foundations for Embedded and Hybrid Systems" has supported, in part, the activities described in this paper.

References

1. J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.
2. Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.
3. "The Model-Driven Architecture", http://www.omg.org/mda/, OMG, Needham, MA, 2002.
4. Grzegorz Rozenberg, "Handbook of Graph Grammars and Computing by Graph Transformation", World Scientific Publishing Co. Pte. Ltd., 1997.
5. Blostein D., Schrr A., "Computing with Graphs and Graph Rewriting", Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.

6. 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.
7. Agrawal A., Karsai G., Shi F.: “Graph Transformations on Domain-Specific Models”, Technical report ISIS-03-403, Vanderbilt University, November, 2003.
8. Vizhanyo A., Agrawal A., Shi F.: “Towards Generation of High-performance Transformations”, Generative Programming and Component Engineering, (submitted), Vancouver, Canada, October 24, 2004.
9. Karsai G., Agrawal A., Shi F., Sprinkle J.: “On the Use of Graph Transformations for the Formal Specification of Model Interpreters”, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.
10. Agrawal A., Simon G., Karsai G.: “Semantic Translation of Simulink/Stateflow models to Hybrid Automata using Graph Transformations”, International Workshop on Graph Transformation and Visual Modeling Techniques, Barcelona, Spain, March 27, 2004.
11. “Simulink Reference”, The Mathworks, Inc., Natick, MA, July 2002. T. A. Henzinger. “The Theory of Hybrid Automata”, In Proc. of IEEE Symposium on Logic in Computer Science (LICS’96), IEEE Press, pp 278-292, 1996.
12. James O. Coplien. Advanced C++: Programming Styles and Idioms. Addison-Wesley, 1992.
13. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley, 1995
14. Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied Addison-Wesley, 2001
15. Using Graph Rewrite Systems for Supporting the Software Design Process by Search for and Substitution of suitable Software Components; Sucrow, B.E., Heverhagen, T.; TAGT ’98 - 6th International Workshop on Theory and Application of Graph Transformation, University of Paderborn, Paderborn, Germany, November 16-20, 1998, Research Report, tr-ri-98-201, pp. 398-406.
16. Angsgar Radermacher. Support for Design Patterns Through Graph Transformation Tools. AGTIVE 1999: pp. 111-126
17. Alexander Christoph. Graph Rewrite Systems for Software Design Transformations; Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002. pp. 76 - 86
18. A. Schurr. PROGRES: A VHL-language based on graph grammars. In in Proc. 4th Int. Workshop on Graph-Grammars and Their Application to Computer Science, number 532 in LNCS, pages 641–659. Springer-Verlag, 1991. T
19. Taentzer, G.: AGG: A Tool Enviroment for Algebraic Graph Transformation, in Proc. of Applications of Graph Transformation with Industrial Relevance, Kerkrade, The Netherlands, LNCS, Springer, 2000.
20. U.Assmann: “Invasive Software Composition”, Springer, 2003.
21. R. Alur, F. Ivancic, J. Kim, I. Lee and O. Sokolsky: “Generating Embedded Software from Hierarchical Hybrid Models”, in Proceeding of ACM SIGPLAN Languages, Compilers, and Tools for Embedded Systems (LCTES’03), San Diego, California, June 11-13, 2003.
22. G. Karsai, S. Neema, B. Abbott, D. Sharp. “A Modeling Language and its Supporting Tools for Avionics Systems”, 21st Digital Avionics Systems Conference, August 2002.
23. Kurt Mehlhorn, “Graph algorithms and NP-completeness”, Springer-Verlag New York, Inc., New York, NY, USA, 1984.