Metamodel Based Model Transformation Language

Aditya Agrawal
Institute for Software Integrated Systems (ISIS)
Vanderbilt University
Nashville, TN – 37235
aditya.agrawal@vanderbilt.edu

ABSTRACT

The Model Driven Architecture (MDA) can have a greater impact by expanding its scope to Domain Specific MDA (DSMDA). DSMDA is the use of MDA for a particular domain. This helps developers to represent their systems using familiar domain concepts. For each DSMDA, a transformer is needed to convert Domain Specific Platform Independent Models (DSPIM –s) to Domain Specific Platform Specific Models (DSPDM-s). Such model transformers are time consuming and error prone to develop and maintain. Hence, a high-level specification language to formally specify the behavior of model transformers is required. The language must also have an execution framework, which can be used to execute the specifications in the language. This research proposes to develop such a language and execution framework that will help to considerably speed-up the development time for model transformers.

Categories & Subject Descriptors: D.2.2 Design Tools and Techniques, F.4.2 [Grammars and Other Rewriting Systems] and D.2.6 [Programming Environments]: graphical environments.

General Terms: Design, Algorithms, Languages.

Keywords: Model Driven Architecture, Model Integrated Computing, Graph grammars and transformations.

1. THE PROBLEM STATEMENT

The MDA [4] effort by OMG has drawn focus to the aims of Model Integrated Computing (MIC) [1]. MIC has benefits in terms of high productivity when applied to specific domains where users are familiar with the use of modeling. To leverage the benefits of MIC in MDA, the MDA scope needs to be expanded to Domain Specific MDA where the focus is on developing the MDA process for specific domains. MIC however, has its own problems such as high development cost, lack of standardization and vendor support **Error! Reference source not found.**.

To tackle these problems, we propose a solution that advocates the development of a framework to support the development and use of Domain Specific Modeling Environments (DSME). This approach helps distribute the cost of the framework to a larger community. It can lead to standardization that will allow vendors to support various domain-specific modeling environments within the framework. A particular DSMDA will consist of a Domain Specific Modeling Environment. This environment is then used to develop Domain Specific Platform Independent Models (DSPIM). These models represent the behavior and structure of the system with no implementation details. Such models then need to be converted to a Domain Specific Platform Specific Models (DSPSM). Theses models may use domain specific libraries and frameworks or they could be domain independent. DSPSM is the general term that covers all the scenarios.

Tools such as GME [2] and DOME [6] already provide a major portion of the framework support. They allow developers to specify the abstract syntax and static semantics of the modeling environments/languages. However, developers spend significant effort in writing code that implements the transformation from Domain Specific Platform Independent Model (DSPIM) to Domain Specific Platform Specific Model (DSPSM).

In order to speed up the development of DSMDAs a high-level specification language is required for the specification of model transformers. An execution framework can then be used to execute specifications expressed in the language. Design of such a language is non-trivial as a model transformer can work with arbitrarily different domains and can perform fairly complex computations.

From a mathematical viewpoint models in MIC are graphs. To be more precise they are vertex and edge labelled multi-graphs. We can then use the mathematical concepts of graph transformations [7] to formally specify the intended behaviour of a model interpreter.

There exists a variety of graph transformation techniques described in [7, 8, 9, 10]. The prominent among these are node replacement grammars, hyperedge replacement grammars, algebraic approaches and programmed graph replacement systems. These techniques have been developed mostly for the specification and recognition of graph languages, and performing transformations within the same "domain" (i.e. graph), while we need a graph transformer that works on two different kinds of graphs. Moreover, these transformation techniques rarely use a widely accepted well-defined language for the specification of structural constraints on the graphs. In summary, the following features are required in the transformation language:

1. The language should provide the user with a way to specify the different graph domains being used. This helps to ensure that graphs/models of a particular domain do not violate the syntax and static semantics of the domain.
2. There should be support for transformations that create independent models/graphs conforming to different domains than the input models/graphs. In the more general case there can be N input model/domain pairs and M output model/domain pairs.
3. The language should have efficient implementations of its programming constructs. The generated implementation should be only a constant factor slower that its equivalent hand written code.

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

4. All the previous points aim to increase productivity and reduction in the time required for writing model interpreters. This is the primary and most important goal.

2. GReAT

The transformation language we have developed to address the needs discussed above is called Graph Rewriting and Transformation language or GReAT for short. This language can be divided into 3 distinct parts: (1) Pattern Specification language, (2) Graph transformation language, and (3) Control flow language.

2.1 Pattern Specification Language

The heart of a graph transformation language is the pattern specification language and pattern matching. The pattern specifications found in graph grammars and transformation languages [7, 8, 9] are not sufficient for our purposes. A more expressive easy to use pattern specification language is introduced that allows specification of complex graph patterns.

The pattern specification language uses a notion of cardinality on each pattern vertex and each edge. The exact semantic meaning of such a construct in terms of pattern matching wasn't immediately obvious. Such patterns have then been associated with unambiguous semantic meaning.

2.2 Rewriting and Transformation Language

In model-interpreters, structural integrity is a bigger concern because model-to-model transformations usually transform models from one domain to models that conform to another domain. This makes the problem two-fold. The first problem is to specify and maintain two different models conforming to two different metamodels (in MIC metamodels are used to specify structural integrity constraints). A greater problem to be addressed is that of maintaining references between the two models. It is important to maintain some sort of reference, link and other intermediate values. These are required to correlate graph objects across the two domains.

The solution to these problems is to use the source and destination metamodels to explicitly specify the temporary vertices and edges. This creates a unified metamodel along with the temporary objects. The advantage of this approach is that we can then treat the source model, destination model and temporary objects as a single graph. Standard graph grammar and transformation techniques can then be used to specify the transformation. The rewriting language uses the pattern language described above. Each pattern object's type conforms to the unified metamodel and only transformations that do not violate the metamodel are allowed. At the end of the transformation, the temporary objects are removed and the two models conform exactly to their respective metamodels. The transformation language is inspired by many previous efforts such as [7, 8, 9, 10].

2.3 Controlled Graph Rewriting and Transformation

There exists a need for a high-level control flow language that can control the application of the productions and allows the user to manage the complexity of the transformation. This prompted us to add a high-level control flow language to GReAT. The control flow language supports the following features:

- Sequencing – rules can be sequenced to fire one after another.
- Non-Determinism – rules can be specified to be executed "in parallel", where the order of firing of the parallel rules is non deterministic.
- Hierarchy – Compound rules can contain other compound rules or primitive rules.
- Recursion – A high level rule can call itself.
- Test/Case – A branching construct used to choose between different control flow paths.

3. ACKNOWLEDGEMENTS

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.

4. REFERENCES

[1] J. Sztipanovits, and G. Karsai, "Model-Integrated Computing", Computer, Apr. 1997, pp. 110-112

[2] A. Ledeczi, et al., "Composing Domain-Specific Design Environments", Computer, Nov. 2001, pp. 44-51.

[3] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.

[4] "The Model-Driven Architecture", http://www.omg.org/mda/, OMG, Needham, MA, 2002.

[5] Agrawal A., Levendovszky T., Sprinkle J., Shi F., Karsai G., "Generative Programming via Graph Transformations in the Model-Driven Architecture", Workshop on Generative Techniques in the Context of Model Driven Architecture, OOPSLA , Nov. 5, 2002, Seattle, WA.

[6] "Dome Guide", Honeywell, Inc. Morris Township, N.J, 1999.

[7] Grzegorz Rozenberg, "Handbook of Graph Grammars and Computing by Graph Transformation", World Scientific Publishing Co. Pte. Ltd., 1997.

[8] Blostein D., Schürr A., "Computing with Graphs and Graph Rewriting", Technical Report AIB 97-8, Fachgruppe Informatik, RWTH Aachen, Germany.

[9] 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.

[10] H. Göttler, "Diagram Editors = Graphs + Attributes + Graph Grammars," International Journal of Man-Machine Studies, Vol 37, No 4, Oct. 1992, pp. 481-502.

[11] Agrawal A., Karsai G., Ledeczi A.: "An End-to-End Domain-Driven Development Framework", Domain Driven Development Track, 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Anaheim, California, October 26, 2003.