An End-to-End Domain-Driven Software Development Framework

Aditya Agrawal Gabor Karsai Akos Ledeczi
Institute for Software Integrated Systems,
Vanderbilt University
Nashville, TN, 37235
{aditya.agrawal, gabor.karsai, akos.ledeczi}@vanderbilt.edu

ABSTRACT

This paper presents a comprehensive, domain-driven framework for software development. It consists of a meta-programmable domain-specific modeling environment and a model transformation generator toolset based on graph transformations. The framework allows the creation of custom, domain-oriented programming environments that support end-user programmability. In addition, the framework could be considered an early, end-to-end implementation of the concepts advocated by the OMG's Model Driven Architecture initiative.

Categories & Subject Descriptors: D.2.2 [Design Tools and Techniques].

General Terms: Design, Algorithms and Languages.

Keywords: Software Development, Model-Driven Architecture, Model-Integrated Computing, Graph Transformations

1. INTRODUCTION

1.1 Classifying Programming Languages

Programming languages can be broadly divided into two categories: (1) General-purpose languages (GPLs), such as assembly, C, C++, Java and (2) Domain-specific languages (DSLs), such as Matlab/Simulink [24]. Tools for general-purpose languages are generally less expensive as a larger community absorbs the cost, whereas DSLs are more expensive, though they can increase productivity by bringing power programming to domain users via familiar specialized notations and languages. It is well know that GPLs have been more prevalent and successful compared to DSLs, even though claims about DSLs' capabilities to increase productivity are widely accepted [26]. The primary reasons behind the limited success of DSLs have historically been the following:

- DSLs are more expensive to create as the development cost and time is borne by a small user community
- Since there is a small user base, tools and support for a DSL is not at par with GPLs and
- The wide user base and longer life of GPLs helps make the language implementations robust and reliable.

Another view of languages divides them into textual and graphical categories. Graphical languages are usually impractical for general-purpose programming but can be useful in a limited context, in specific domains. One of the most successful recent examples of graphical, domain-specific languages is Matlab/Simulink [24] for simulation and control engineering. We believe that a mixed textual and graphical notation can be helpful in limited domains. For example, in the software development domain, the UML [3] specification has both textual (Object Constraint Language) and graphical (Use-Case Diagram, Class Diagram, etc.) notations. In hardware development domain, tool vendors [27] are now providing a graphical notation for the structural description of hardware while the behavioral description is still textual.

For DSLs to become more popular the three hurdles mentioned above must be addressed. A key limitation is the *cost of development* (in terms of time and effort), which we conjecture can be reduced by creating a framework for developing DSLs. This approach has several advantages. First, the framework can be used to develop many languages, and thus the cost and time of development is reduced and can be absorbed by a larger community. Second, the framework can be the focal point for a wider user base, thus making it profitable for industries to provide support and tools. Within the framework there will be a development cost for a given DSL that needs to be minimized for the framework to be effective.

1.2 Requirements of a Domain-Driven Software Development Framework

A useful framework for developing domain-specific graphical languages should have a basic set of features. The features can be divided into the following two categories:

- *Meta framework* tools that will be used to describe the syntax, semantics and visualization of DSLs. The meta framework must provide support for the specification of a language defined by its abstract syntax, concrete syntax, static semantics, dynamic semantics, and visualization. The syntax of a programming language describes the structure of programs without consideration of their meaning. The abstract syntax of the language captures the abstract concepts and their relationships used in the language. Issues such as type-compatibility are captured in the static semantics of the language. Dynamic semantics is defined as the relation of the abstract syntax to a model of computation. In other words, it can be considered as a mapping from one language to another (provided the model of computation is captured in a linguistic framework).
- *Language framework* tools that will be used for the creation, visualization and verification of sentences in a domain-specific language. The language framework should allow the use of the language in an integrated environment that includes creating, editing, and deleting sentences of the

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

language; visualization of the sentences; etc. Apart from editing of the sentences, the framework needs to enforce the concrete syntax and static semantics of the language using some mechanisms. The final requirement of the framework is to be able to use transformation tools that map sentences of the language into sentences of some model of computation [20]. Examples of such models of computation are stack machines, process networks, finite state machines, etc. Often, although not always, sentences expressed in target the model of computation are executable, hence they are called "executable models".

1.3 A Domain-Driven Development Framework

The key components of the framework are the following:

- *Generic Modeling Environment* (GME) [2],
- GME's metamodeling language [2],
- *Graph Rewriting and Transformation* (GReAT): a model transformation specification language and
- *GReAT Execution Engine* (GReAT-E): an execution framework for GReAT specifications.

Figure 1 shows how these different technologies fit together to form an end-to-end framework for domain-driven software development. The metamodeling language is used for the specification of syntax static semantics and visualization of a DSL. The metamodel transformer can convert this specification (a metamodel) into an internal representation. This internal representation is then used to configure GME to support the specified domain-specific language. GReAT is a graphical language used to specify the semantics for the DSL. GReAT-E can execute GReAT specifications on sentences of the DSL to produce executable models.

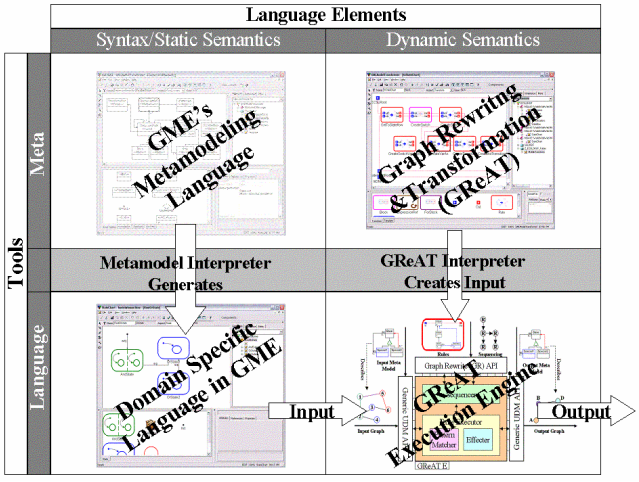


Figure 1 A Domain-Driven Development Framework

1.4 Paper Organization

This paper showcases a domain-driven software development framework and demonstrates how it achieves the goals outlined in Section 1.2. The remainder of this paper is organized as follows: Section 2 describes the meta framework and how GME itself was used to develop the meta framework; Section 3 describes the language framework; Section 4 presents an example that illustrates the specification of a simple language; Section 5 presents some preliminary results; Section 6 discusses the conclusions and proposals for future research.

2. THE META FRAMEWORK

2.1 Infrastructure: The Generic Modeling Environment (GME)

The domain-driven development framework has been developed around GME, a meta-programmable tool that can be customized to support various visual domain-specific modeling languages. GME uses direct manipulation techniques for editing complex models. The direct manipulation front-end is comprised of the "editing engine" that operates on data structures representing the domain-specific models. How these data structures are organized, what objects and relationships are allowed, what attributes objects have, etc. are captured in an internal metamodel. The internal metamodel is a read-only, static data structure that GME uses at run-time to connect the model data structures to the visualization and direct manipulation tools. It connects to a database backend to provide persistence services and to validate the editing operations. This internal metamodel is created from an external metamodel, which is built by the designer of the domain-specific modeling language [2].

The external metamodel is a sentence in a visual metamodeling language. GME was used to create this metamodeling language. Sentences of this language (metamodels) can be created and edited in GME. The metamodeling language is described in Section 2.2. GME also provides a special model transformation tool called metamodel transformer that transforms an external metamodel into an internal metamodel [2].

2.2 Abstract and Concrete Syntax

The external metamodel captures the abstract and concrete syntax of the modeling language supported by the GME. The abstract syntax is expressed in the form of UML class diagrams [3] that introduce domain concepts (classes), their attributes, and their relationships. These UML class diagrams describe all the possible models that can be built via the modeling language, similar to how Extended Backus-Naur Form (EBNF) describes all the possible sentences in a textual language. In other words, we use UML class diagrams to represent generative grammars for models.

The concrete syntax is expressed by coupling the domain modeling entities (classes and associations) to specific visualization features available in GME. This coupling happens in two ways:

- Using specific stereotypes for classes. GME defines a set of stereotypes, and when these are assigned to classes in the metamodel they determine how the visualization should happen. For example, <<Model>>-s are visualized as containers shown as icons with connection ports, <<Atom>>-s as (user-defined) icons, <<Connection>>-s as lines, etc.
- Using specific idioms in the metamodel. GME's metamodeling tool uses a number of predefined idioms: patterns over classes that carry special meaning. For example, a <<Reference>> class associated with an <<Atom>> class via an association labeled as "refersTo"

means that (in a context) one can use reference objects that point to atomic objects of the selected kind.

The GME has a number of visualization techniques (e.g., container objects), and there is a set of well-defined stereotypes that allow relating the domain entities to GME visualization concepts. Some visualization techniques require multiple, cooperating domain classes, and in this case specific idioms are used in the GME. For details, please see the GME documentation [2].

2.3 Static Semantics

The GME metamodels discussed above only allow specification of the abstract syntax, i.e., they do not support semantic constraints on the models. These constraints define the well-formedness rules for the models, i.e., the static semantics. We use OCL [19] to express these rules in the GME metamodels. One can couple OCL expressions to model elements, and the GME editing engine evaluates these expressions at run-time. If constraint violations are found, the user is warned about the specific rule that has been violated. Notification of constraint violation during model editing can be annoying, thus there exists a fine-grain control over the evaluation time of constraints.

2.4 Semantics via Transformation Specification

Most textual languages have either (1) a direct one-to-one mapping from the source to the target model of computation or (2) have no formal specification for the transformation. The transformation specification is buried in the code generator. The process of writing generators is time consuming and costly and therefore, a better approach is necessary to support a domain-driven software development framework. If a higher-level specification for the model transformations is available, it can presumably be used to generate the code for the model translator. From a mathematical viewpoint, one can recognize that domain-specific models are graphs or to be more precise: vertex and edge labeled multi-graphs, where the labels are denoting the corresponding entities (i.e., types) in the metamodel. Thus, the model transformation problem can be converted into a graph transformation problem. We can then use the mathematical concepts of graph transformations to formally specify the intended behavior of model transformers.

Graph grammars and graph transformations (GGT) have been recognized [11][12][13][14] as a powerful technique for specifying complex transformations that can be used in different places in a software development process. Many tasks in software development can be formulated using this approach, including weaving of aspect-oriented programs [23], application of design patterns [13], and the transformation of platform-independent models into platform specific models [4].

A variety of graph transformation techniques are described in [5][6][7][8][9][10][17]. These techniques include node replacement grammars, hyperedge replacement grammars, algebraic approaches, and programmed graph replacement systems. Most of these techniques have been developed for specifying and recognizing graph languages and performing transformations within the same "domain" (i.e., graph) however, while we need a graph transformer that works on two different kinds of graphs. Moreover, these transformation techniques rarely use a well-defined language for the specification structural constraints on the graphs. In summary, the following features are required in the transformation language:

- 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.
- 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.
- The language should have efficient implementations of its programming constructs. The generated implementation should be comparable to its equivalent hand written code.
- All the previous points aim to increase productivity and achieve speed up in the time required for writing model interpreters. This is the primary goal.

2.5 Language for Graph Rewriting and Transformations

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

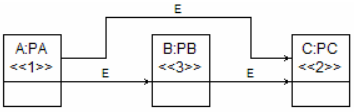
2.5.1 The Pattern Specification Language

The heart of a graph transformation language is the pattern specification language and the related pattern matching algorithms. The pattern specifications found in graph grammars and transformation languages [5][6][7][8][15][16][17][18] are not sufficient for our purposes, as they do not follow UML concepts. This paper briefly introduces an expressive – yet easy to use – pattern specification language that is tightly coupled to the UML class diagrams. String matching will be used to illustrate representative analogies.

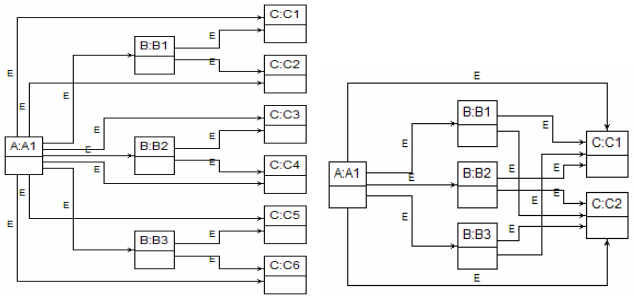
Patterns in most graph transformation languages have a one-to-one correspondence with the host graph.. Consider an example from the domain of textual languages where a string to match starts with an 's' and is followed by 5 'o's. To specify such a pattern string we could enumerate the 'o's and write "soooooo". Since this is not a scalable solution, a representation format is required to specify such strings in a concise and scalable manner. One can use regular expressions: for strings we could write it as "s5o" and use the semantic meaning that o needs to be enumerated 5 times. The same argument holds for graphs, and a similar technique can be used. Cardinality can be specified for each pattern vertex with the semantic meaning that a pattern vertex must match *n* host graph vertices, where *n* is its cardinality. However, it is not obvious how the notion of cardinality truly extends to graphs. In text, we have the advantage of a strict ordering from left to right, whereas graphs do not possess this property.

In Figure 2 (a) we see a pattern having three vertices. One possible meaning could be *tree semantics*, i.e., if a pattern vertex pv1 with cardinality c1 is adjacent to pattern vertex pv2 with cardinality c2, then the semantics are that each vertex bound to v1 will be adjacent to c2 vertices bound to v2. These semantics

when applied to the pattern gives Figure 2 (b). The tree semantic is weak in the sense that it will yield different results for different traversals of the pattern vertices and edges and hence, it is not suitable for our purpose.



(a) Pattern with three vertices



(b) Tree semantics (c) Set semantics

Figure 2 Pattern with different semantic meanings

Another possible unambiguous meaning could use *set semantics*: consider each pattern vertex pv to match a set of host vertices equal to the cardinality of the vertex. Then an edge between two pattern vertices pv1 & pv2 implies that in a match each v1, v2 pair should be adjacent, where v1 is bound to pv1 and v2 is bound to pv2. This semantic when applied to the pattern in Figure 2 (a) gives the graph in Figure 2 (c). The set semantics will always return a match of the structure shown in Figure 2 (c), and it does not depend upon factors such as the starting point of the search and how the search is conducted.

Due to these reasons, we use set semantics in GReAT and have developed pattern-matching algorithms for both single cardinality and fixed cardinality of vertices.

2.5.2 Graph Rewriting Transformation Language

Pattern specification is just one important part of any graph transformation language. Other important concerns are the specification of static structural constraint in graphs and ensuring that these are maintained throughout the transformations [6]. These problems have been addressed in a number of other approaches, such as [15][16].

In model-transformers, structural integrity is a primary concern. Model-to-model transformations usually transform models from one domain to models that conform to another domain making 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). An even more important problem to address involves maintaining references between the two models. For example, it is important to maintain some sort of references, links, and other intermediate values required to correlate graph objects across the two domains.

Our solution to these problems is to use the source and destination metamodels to explicitly specify the temporary vertices and edges. This approach 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. Our transformation language is inspired by many previous efforts, such as [7][8][9][17][18].

The graph transformation language of GReAT defines a *production* (also referred to as rule) as the basic transformation entity. A *production* contains a pattern graph that consists of pattern vertices and edges. These pattern objects conform to a type from the metamodel. Each pattern has another attribute that specifies the role it plays in the transformation. A pattern can play the following three different roles:

1. *Bind* – used to match objects in the graph.
2. *Delete* – also used to match objects in the graph, but after these objects are matched they are deleted from the graph.
3. *New* – used to create objects after the pattern is matched

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 from the match and objects marked *new* are created.

Sometimes the patterns by themselves are not enough to specify the exact graph parts to match and we need other, non-structural constraints on the pattern. An example for such a constraint is: "the value of an attribute of a particular vertex should be within some limits." These constraints or pre-conditions are expressed in a *guard* and are described using Object Constraint Language (OCL) [19]. There is also a need to provide values to attributes of newly created objects and/or modify attributes of existing object. *Attribute Mapping* is another ingredient of the production: it describes how the attributes of the "new" objects should be computed from the attributes of the objects participating in the match. Attribute mapping is applied to each match after the structural changes are completed.

A production is thus a 4-tuple, containing a pattern graph, mapping function that maps pattern objects to actions, a guard expression (in OCL), and an attribute mapping.

2.5.3 Controlled Graph Rewriting and Transformation

To increase the efficiency and effectiveness of GReAT, it is essential to have efficient implementations for the productions. Since the pattern matcher is the most time consuming operation, it needs to be optimized. One solution is to reduce the search space (and thus time) by starting the pattern-matching algorithm with an initial context. An initial context is a partial binding of pattern objects to input (host) graph objects. This approach significantly reduces the time complexity of the search by limiting the search space. In order to provide initial bindings, the production definition is expanded to include the concept of ports. Ports are elements of a production that are visible at a higher-level and can then be used to supply initial bindings. Ports are also used to retrieve output objects from the production.

The next concern is the application order of the productions. In graph grammars there is no ordering imposed on productions. If the pattern to be matched exists in the host graph and if the pre-condition is met then the production will be executed. Although this technique is useful for generating and matching languages, they are unsuitable for model-to-model transformations that are algorithmic in nature and require strict control over the execution sequence. Moreover, a well-defined execution sequence can be used to make the implementation more efficient.

There is a need for a high-level control flow language that can control the application of the productions and allow the user to manage the complexity of the transformation. The control flow language of GReAT supports the following features:

- Sequencing – rules (in GReAT the productions are called rules) can be sequenced to fire one after another. This is achieved by attaching the output port of the first rule to the input port of the next rule.
- Non-Determinism – when required parallel execution of a set of rules can be specified. The order of execution of these rules is non-deterministic. This construct is achieved in GReAT by attaching the output of one rule to the input of more than one rule.
- Hierarchy – High-level rules have been introduced in the language. These are used for encapsulation and data abstraction. Compound rules can contain other compound rules or primitive transformation rules.
- Recursion – A high level rule can "call" itself.
- Test/Case – A conditional branching construct that can be use to choose between different control flow paths.

3. THE LANGUAGE FRAMEWORK

3.1 Infrastructure: The Generic Modeling Environment (GME)

As noted above, the same GME is used to support domain modeling. However, this GME instance is configured by the internal metamodel to support and enforce the specific features of the domain-specific modeling language. A domain-specific instance of GME provides a tool with domain-oriented features and symbols for model editing and manipulation. Its capabilities have been discussed in detail elsewhere [2].

3.2 Run-time support for model transformations

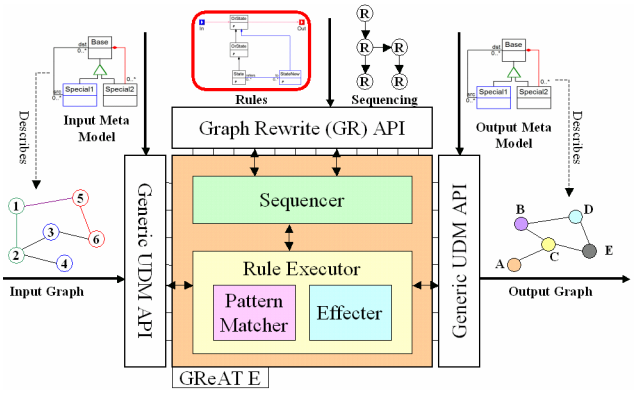


Figure 3 The GReAT Interpreter

The model transformation language described above is supported through a *Graph Rewiring and Transformation Execution Engine* (GReAT-E). Figure 3 shows its architecture. The engine works as an interpreter: it takes the model transformation "program" in the form of a data structure, and it "executes" it on an input graph to produce an output graph. The engine uses generic API-s (using our model-driven reflection package called UDM [21]), and is thus suitable for executing any model transformation. Work is currently underway to translate the model transformation specifications into code that can be executed directly.

4. DEVELOPING A SIMPLE LANGUAGE

This section develops an example language to demonstrate the capabilities of our end-to-end framework. We will call the language being developed Modeling Language for Embedded Systems (MOLES). The embedded system community develops a large class of applications that are event driven and use the data flow semantics. To create these applications, developers must first build the basic data flow components and then connect them together in different configurations to achieve the desired application. The requirements of the MOLES language therefore involves the following capabilities:

- Design of component interface and behavior
- Creating data flow graphs using the components
- Facility to have timer interrupts, queues and delay elements.

4.1 Language Specification

The first step in the development of a new language in the GME is to specify the syntax and the visualization using the metamodeling environment. The metamodel for MOLES is divided into two parts; the first describes the internals of a component, while the second deals with developing packages or applications based on the components.

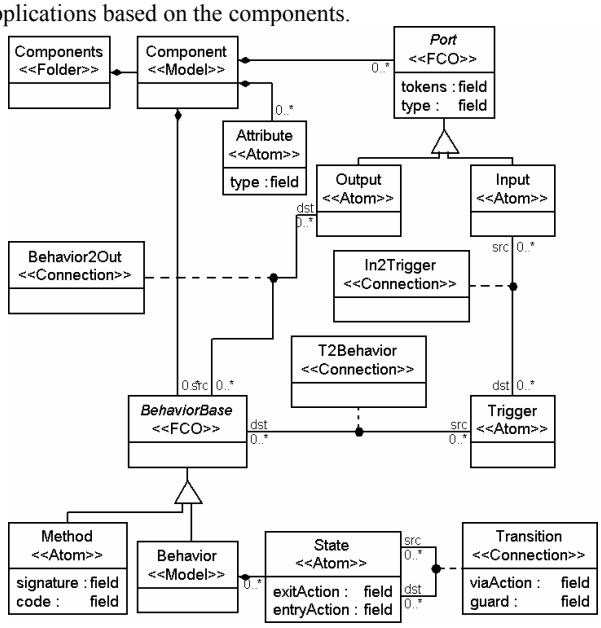


Figure 4 Definition of a component

Figure 4 shows the metamodel for components. Here, we can see that a *Component* can contain *Ports*, *Behaviors*, and *Attributes*.

Based upon whether they receive or send data the *Ports* are specialized to be either *Input* or *Output*. *Behavior* can be specified in two different ways: the first is to specify a piece of code that implements the behavior, whereas the second is to describe the behavior using a simple state machine. *Attributes* capture data storage elements of the *Component*.

Figure 5 shows how these components can be put together in a package. A package can contain components and component references; they also contain ports and intermediate delay elements. An important thing to note is that the *Component* and *Port* are the same as in Figure 4. *Package* and *Component* have the same kind of ports and hence the connection between *ConnectionBase* elements defines all the possible dataflow connections. *Package* can contain other *Packages*; this can be used for hierarchical decomposition of the dataflow. *Ports* can not only connect to other *Ports* but also connect to *Delays* and *Timers*. A *Delay* element in a dataflow path introduces a delay of one time step. This means that the data passing through a delay will be held for one time step before to moves forward. *Timers* generate data at periodic intervals.

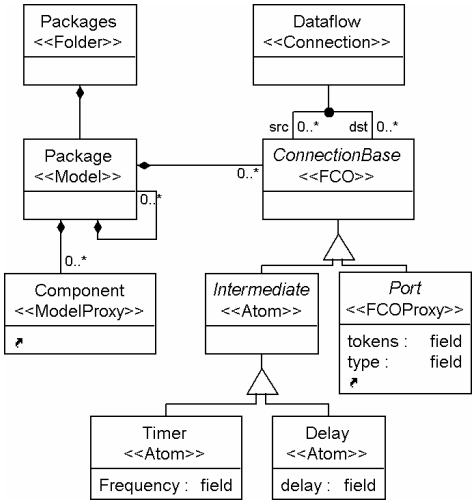


Figure 5 Package definition and dataflow

After defining the syntax, we can define static semantics using OCL constraints. For example, a timer component cannot be the destination of a data flow component. Such a constraint can be specified in OCL and attached to the *timer* (as shown in Figure 6). These constraints are checked by GME when the user is creating domain models. This guarantees that the models will not violate the static semantics.

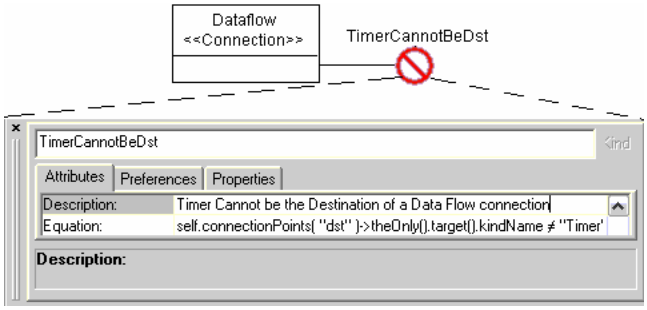


Figure 6 OCL constraint

4.2 Transformation Specification

After the abstract syntax, concrete syntax, and static semantics have been defined, the next step is to define the semantics of the language using GReAT. To define a transformation, we first identify a mapping from the language to the appropriate model of computation. We have chosen synchronous data flow [25]. In synchronous data flow the number of tokens consumed and produced on each port of a component is fixed and predefined. For the sake of brevity we have fixed the token size on each *Port* to be one. Synchronous data flow is a simple and efficient model of computation, and an algorithm exists that can compute the static schedule for node invocations for any synchronous dataflow network [25]. If there is a cyclic dependency in the data flow, the algorithm can also report such problem. A cycle in MOLES that contains a delay doesn't represent a cyclic data dependency assuming the delay component is initialized with a token. A delay with a dataflow connection to an input port of a component signifies that the input port has an initial token and thus can run once without requiring another token on that port. Since every dataflow path will have one token each after one cycle, the component will never be short of tokens on the delay edge. Thus the delay edge can be ignored for solving the scheduling problem. In order to find a static schedule for the dataflow components a topological sort needs to be performed. The topological sort will produce a scheduling order and if a cycle exists it will fail and report that a cycle was found. The target model of computation can be considered a line graph that represents one periodic admissible sequential schedule (PASS) [25].

The transformation rules are shown in Figure 7. At the top level, we have the *TopologicalSort* and *Success?* Rules. *TopologicalSort* encapsulates the sorting algorithm while *Success?* Is used to check if the sort was successful or if it detected a cycle. *TopologicalSort* consists of series of simple transformation rules. The component with no incoming dataflow connections is chosen; this component is added to the PASS and deleted from the input (deletion can be performed on a local copy of the input model). This process is continued until we have exhausted all the components in the input or we find a cycle.

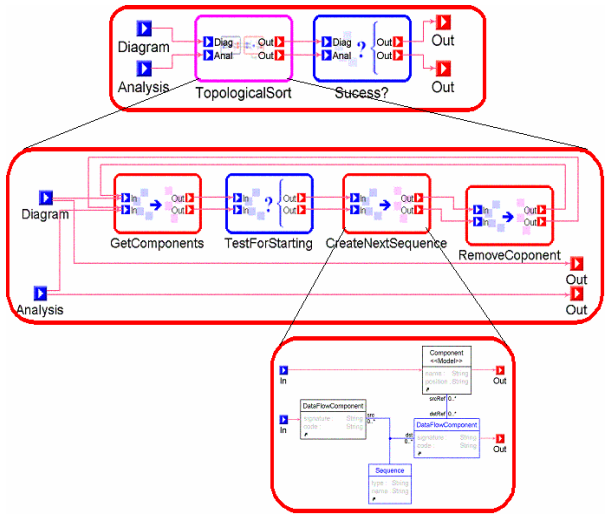


Figure 7 Snapshot of the rules for performing the topological sort

The transformation specification can then be executed using GReAT-E on any MOLES model to generate a PASS.

4.3 Using the language

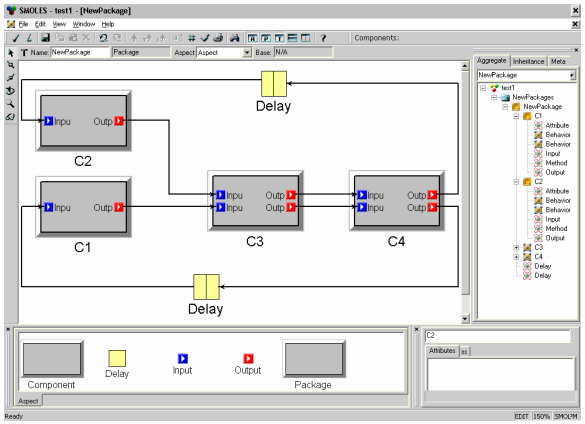


Figure 8 Example models in the MOLES Language

We have seen in previous sections that GME can be used as the editor of different domain-specific languages. It can be used to specify the syntax and semantics of the language using the metamodeling language, as well as used to create sentences of a domain-specified language. The user can use GME to create and edit models that belong to MOLES, where MOLES is the language we just developed. Data flow graphs created in MOLES can then be used to generate a static schedule or find out if there are cycles in the graph.

Figure 8 shows an example dataflow graph developed in the MOLES language using GME. GME provides editing, visualization, syntax and static semantic checking, and safety of the language. For example, the constraint specified in the metamodel of MOLES about timer being only source is enforced by GME in Figure 9.

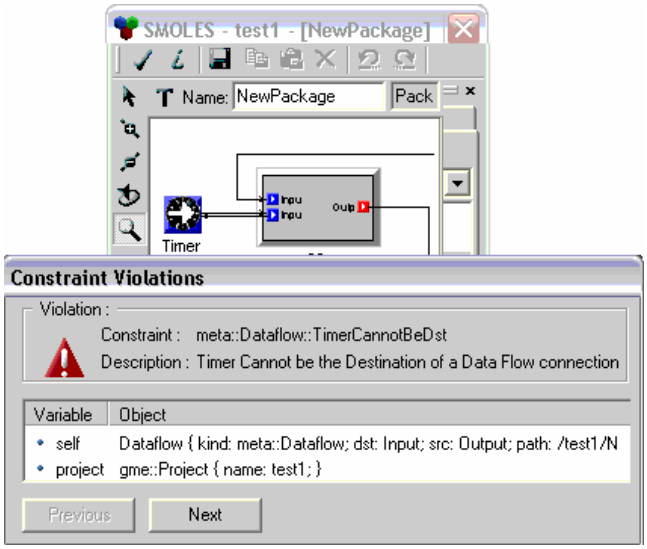


Figure 9 An OCL constraint violation message box

5. PRELIMINARY RESULTS

The goal of using a domain-specific development process is to increase the domain programmer's productivity. The goal of using a metamodel driven development process to create domain-specific development tools is to increase the productivity of the tool developer. Table 1 shows some preliminary results by comparing the size of and time taken to develop GReAT specifications for model transformation problems to estimated equivalent lines of procedural code. The primitive rules are rules that contain graph transformation specification while compound rules are higher-level control flow constructs. Some preliminary tests have shown that each primitive rule corresponds to approximately 30 lines of hand code. The corresponding hand code is fairly complex and not very natural to write. This makes us believe that the language can actually provide increase in productivity. However, better tests need to be designed and performed using more subjects to provide more precise results.

Table 1: Comparison of GReAT specification VS hand code

Problem	GReAT		Hand code
	Primitive/Compound Rules	Time (man-hours)	Est. LOC
Mark and sweep algorithm on Finite State Machine (FSM)	7/2	~2	100
Hierarchical Data Flow (HDF) to Flat Data Flow (FDF)	11/3	~3	200
Hierarchical Concurrent State Machine (HCSM) to Finite State Machine (FSM)	21/5	~8	500
Matlab Simulink/ Stateflow to Hybrid System	66/43	~20	3000

6. CONCLUSIONS AND FUTURE WORK

This paper has identified three historical limitations of domain-specific languages (DSLs) and proposed a framework-based solution to counter then. The dominant reasons for the limitations of previous generations of DSLs are development cost and lack of tool support. Using a framework approach to domain-driven development, we can push the development cost to a one-time investment in a framework and allow independent vendors to provide greater support for the framework. Such a framework needs to satisfy different criteria (such as low cost of development of end-to-end domain languages and good tool support) at both the metamodel and language level. The framework presented in the paper that comprises of Generic Modeling Environment (GME), the metamodeling language, Graph Rewriting and Transformation (GReAT) and GReAT Execution Engine (GReAT-E) has the required capabilities.

In the framework, the abstract and concrete syntax of a language is captured using a UML-based approach called metamodeling. Likewise, static semantics are captured with the help of OCL expressions and the semantics of the language are captured with the help of a transformation specification in GReAT. The syntax, semantics, and transformations are converted to an executable form using the metamodeling transformer, OCL expression checker, and the GReAT interpreter. At the language level, GME provides the user with editing, visualization, and consistency checking and GReAT is used to convert models to executable models of an appropriate model of computation.

This paper illustrated the capabilities of the GME based framework using a simplified example. We have developed and tested a variety of small-to-medium-sized languages using this approach. Preliminary results demonstrate a speedup in the development time.

There are a number of open questions that we plan to address in our ongoing research. Although we have successfully tackled two problems plaguing the domain-driven development community, we still must address the issue concerning language robustness. This issue can be approached if we can reason about the languages built in the framework and guarantee soundness properties. We envision that we can construct languages that are correct by construction, thereby ensuring robustness and correctness.

7. 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. Tihamer Levendovszky and Jonathan Sprinkle have contributed to the discussions and work that lead to GreAT, and Feng Shi has written the first implementation of GreAT-E. The authors would like to thank the reviewer for their valuable comments.

8. 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] 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.

[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] 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.

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

[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] D. Blostein, H. Fahmy, and A. Grbavec, “Practical Use of Graph Rewriting”, 5th Workshop on Graph Grammars and Their Application To Computer Science, Lecture Notes in Computer Science, Heidelberg, 1995.

[11] U. Assmann, “How to Uniformly specify Program Analysis and Transformation”, Proceedings of the 6 International Conference on Compiler Construction (CC) '96, LNCS 1060, Springer, 1996.

[12] A. Maggiolo-Schettini, A. Peron, “A Graph Rewriting Framework for Statecharts Semantics”, Proc.\ 5th Int.\ Workshop on Graph Grammars and their Application to Computer Science, 1996.

[13] A. Radermacher, ``Support for Design Patterns through Graph Transformation Tools", Applications of Graph Transformation with Industrial Relevance, Monastery Rolduc, Kerkrade, The Netherlands, Sep. 1999.

[14] A. Bredenfeld, R. Camposano, “Tool integration and construction using generated graph-based design representations”, Proceedings of the 32nd ACM/IEEE conference on Design automation conference, p.94-99, June 12-16, 1995, San Francisco, CA.

[15] H. Fahmy, B. Blostein, “A Graph Grammar for Recognition of Music Notation”, Machine Vision and Applications, Vol. 6, No. 2 (1993), 83-99.

[16] 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).

[17] G.Schmidt, R. Berghammer (eds.), “Proc. Int. Workshop on Graph-Theoritic Concepts in Computer Science”, (WG ’91), LNCS 570, Springer Verlag (1991).

[18] H.Ehrig, M. Pfender, H. J. Schneider, “Graph-grammars: an algebraic approach”, Proceedings IEEE Conference on Automata and Switching Theory, pages 167-180 (1973).

[19] Object Management Group, Object Constraint Language Specification, OMG Document formal/01-9-77. September 2001.

[20] Edward A. Lee, "Embedded Software," Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

[21] A. Bakay, “The UDM Framework,” http://www.isis.vanderbilt.edu/Projects/mobies/.

[22] J. McCarthy “Recursive functions of symbolic expressions and their computation by machine – I”, Communications of the ACM, 3(1), 184-195, 1960.

[23] Uwe Assmann, “Aspect Weaving by Graph Rewriting”, Generative Component-based Software Engineering (GCSE), p. 24-36, Oct 1999.

[24] “Simulink Reference”, The Mathworks, Inc., July 2002.

[25] E. Lee, D. G. Messerschmitt, “Static Scheduling of Synchronous Dataflow programs for Digital Signal Processing”, IEEE Transactions on Computers 36(1): 24-35, 1987.

[26] J. Gray, G. Karsai, “An Examination of DSLs for Concisely Representing Model Traversals and Transformations”, 36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, p. 325a, January 06 - 09, 2003.

[27] ActiveHDL, http://www.aldec.com/ActiveHDL/, Aldec Inc., Henderson, NV 89074.