Model-Driven Software Development of Model-Driven Tools: A Visually-Specified Code Generator for Simulink/Stateflow

Attila Vizhanyo, Sandeep Neema, Zsolt Kalmar, Feng Shi, and Gabor Karsai
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract

On one hand, visual modeling languages are often used today in engineering domains, Mathworks' Simulink/Stateflow for simulation, signal processing and controls being the prime example. On the other hand, they are also becoming suitable for implementing other computational tasks that assist in model-driven development, like model transformations. In this paper we briefly introduce GReAT: a visual modeling language with simple, yet powerful semantics for implementing transformations on attributed, typed hypergraphs with the help of explicitly sequenced graph transformation rules. The main contribution of the paper is the "highlights" of a specialized model transformation tool: a code generator that generates executable code from the input Simulink/Stateflow models.

1. Introduction

Visual programming still has to make its way into mainstream programming. However, with the arrival of model-based development practices, visual approaches made inroads into the mainstream software engineering - albeit in a somewhat different way. The Unified Modeling Language (UML) is often used today to specify and design systems, and used occasionally for actually implementing systems through code generation. The Model-Driven Architecture (MDA) vision of OMG advocates the use of model transformations in the development process, where these transformations could (also) be visually specified.

We conjecture that the use of visual languages and visually specified transformations are especially relevant for domain-specific modeling languages (DSML) and transformations on those languages — and thus can be a crucial part of a model-driven software development process.

In the work described below the application domain is control engineering and signal processing, where controllers are "signal processors" that are ultimately implemented in software. The flagship commercial product that supports this domain is Mathworks' Simulink/Stateflow (MSS). This language allows specification of embedded controllers in terms of block-oriented signal-flow diagrams (to represent "processing") and Statecharts (to represent "discrete control").

We have developed a model transformation technology in a tool (GReAT), which is based on graph transformation principles. GReAT uses a visual notation for describing the model transformations in terms of explicitly-sequenced transformation rules. We argue that GReAT, being a DSML for model transformations, increases agility through reducing complexity: it makes easier the task of engineers who build model transformations. Raising the level of abstraction in the domain of model transformations means that developers can specify transformations in terms of abstractions of the target domain(s), not in the solution domain. GReAT has well-defined execution semantics, which narrows the abstraction gap that developers needs to bridge between the problem and the solution domains.

We were interested in trying out this technology for implementing a non-trivial model transformation program: a code generator for MSS that generates C code from the high-level models for execution on a distributed embedded platform. In this paper we briefly introduce GReAT, describe the code generator, and then evaluate the results.

2. Background: The GReAT Approach

The graph transformation based language we have developed is called GReAT, short for "Graph Rewriting and Transformation" [1]. GReAT is suitable for formal specification of model transformations, where UML class diagrams are used to represent the abstract syntax of the input and the output models of the transformation. The models are represented with *vertex and*

edge labeled multi-graphs, where the labels are denoting the corresponding types in UML class diagrams. The transformations are represented as explicitly sequenced elementary rewriting operations, called productions or rules. A production contains a pattern graph that consists of pattern vertices and edges, jointly called pattern elements. Each pattern element has an attribute, called the role that specifies what happens during the transformation step. A pattern element can play one of three roles: *Bind*, *Delete*, or *New*. The execution of a rule involves matching every pattern object marked with either *Bind* or *Delete*. The pattern matcher will return all possible matches for the given pattern and a host graph. An (optional) guard condition can filter out undesirable matches from this set. Then for each match the pattern objects marked *Delete* are deleted from the graph and objects marked *New* are created. Finally, the attributes of the newly created graph objects can be updated by an optional `AttributeMapping` (AM) code, written in an executable language (C++).

GReAT uses the UML class diagram notation for the specification of patterns. For example, in Figure 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 *New* role. The semantics of the rule is: find the pattern that matches the elements with the *Bind* role, 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 *New*, in this case `NewState`, `OrState→NewState` composition and `State↔NewState` association. Finally, use `AttributeMapping` to fill in the attributes of the newly created objects.

The computational complexity of the pattern matching can be significantly reduced if we provide the pattern matcher with an initial context. By "context" we mean an initial partial match that is give to the pattern matcher when it is started. The initial matches 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 1 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.

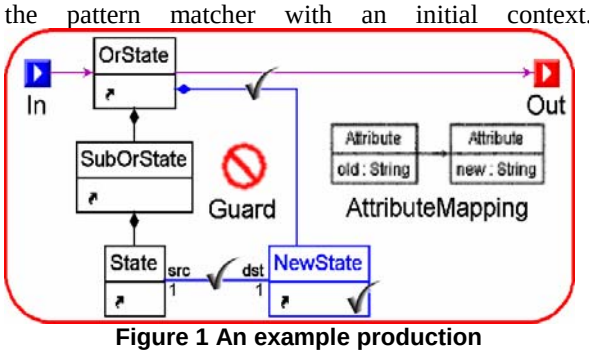

Figure 1 An example production

In order to better manage complexity in transformation programs it is important to have higher-level constructs, like hierarchical rules and control structures in the graph rewriting language. For this reason, we support (1) the nesting of rules and (2) control structures. Figure 2 shows how an order of execution can be specified using control structures in GReAT. In this example the rules are executed from left to right starting from *CreateRootFunction*. GReAT supports hierarchical nesting 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*, giving the user control over the traversal strategies over the host graph. Compound rules *CreateInputArgs* and *CreateOutputArgs* shown in Figure 2 represent *Blocks*, that contain other compound and primitive rules. 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.

The GReAT language is implemented using the (1) GR Engine, and the (2) GR Code Generator tool. The GR Engine is a generic "interpreter" that takes the input graph, applies the model transformation to it, and thus generates the output graph. The GR Code Generator tool generates efficient C++ code that is specific to a particular transformation. Then the generated code is compiled into a binary executable that is used to perform the transformation with significant performance gains over the GR Engine.

We conjecture that using GReAT to develop model transformers for an MDSD process has two primary benefits: (1) the semantics of the GReAT language allows for decreased development time in an easy-to-use visual programming environment, (2) the semantic

Figure 2 An example control structure

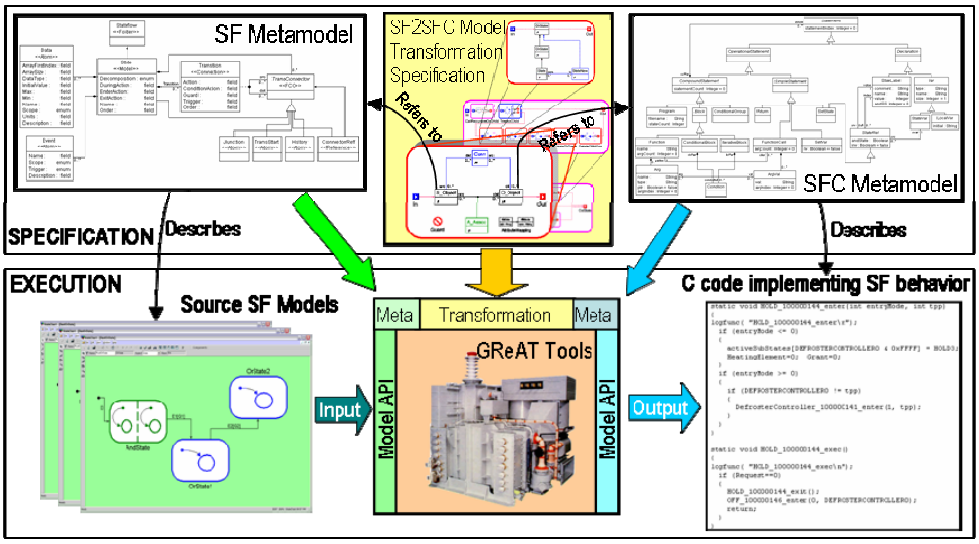

Figure 3: Model-driven construction of a model transformation tool: a code generator

mapping between the input and the output of the transformation becomes formal and explicit, which allows a better understanding of what the transformation does.

To illustrate these benefits, we briefly describe a specialized model transformation: a code generator that was specified using GReAT. This transformation generates C code from MSS models [2]. Figure 3 illustrates the process: on the top the metamodels for the input and the output and the models for the transformation are shown. The input models comply with the input metamodel, while the generated C code complies with the output, as shown on the bottom. The transformation models are "operationalized" via the GReAT tools (i.e. the GR engine, or the executable code produced by the code generator).

3. The Stateflow to C Code Generator

Controllers modeled using Stateflow are often used in safety-critical applications such as those in "x-by-wire"[1]. The correctness and verifiability of code-generators thus becomes an important consideration. The declarative nature of the specification in GReAT opens up the possibility of verification. Our work is also driven by the specifications of Mathworks Automotive Advisory Board (MAAB) that defines restrictions on the use of Stateflow constructs in modeling an automotive application, in effect defining a subset of Stateflow that can be used in automotive applications. A design goal and an additional benefit of specifying code-generators with GReAT, is the explication of semantics of Stateflow.

The C code that must be generated from the diagrams to implement the operational semantics of Stateflow, is a stylized subset of C. This stylization is driven by the fact that the code for implementing state-based behavior follows some specific idioms and patterns. For this subset we have created a metamodel in UML, that we call SFC (an abbreviation of Stateflow-C). Some of the key constructs of SFC metamodel include: `Statement`, `Declaration`, `Operational-Statement`,`Functions`, `FunctionCall`, `StateVar`, `SetState`, `CheckState`, among others. Please note that the SFC metamodel is an important contribution of this work, as it leads to a cleaner semantic underpinning of Stateflow by explicitly formulating the abstract machine for executing Stateflow (in terms of C code fragments).

Mathworks Stateflow operational semantics is defined in the online help documentation included with the Stateflow release [3]. The semantics is described in the form of informal execution rules for various scenarios, such as executing a chart, executing a state, entering a state, executing a flow graph, etc. The code that our code generator produces is in effect a partial specialization (specialized for the input Stateflow model) and instantiation of these rules.

In the generated code, we produce enter, exit, and exec (or step) function for each state in the Stateflow. The core of the state machine behavior is encoded in these functions. The enter function performs the actions that are required to activate a state. The Mathworks Stateflow document specifies 7 different steps involved in activating a state [3]. These steps can be partitioned in 4 non-exclusive groups. Depending on the mode of entry different groups of steps must be performed, e.g.: if a state is entered whose parent state is not active (typically in case of cross-hierarchy transition) then the execution semantics require execution of steps 1-4 for the parent state. In our generated code,

[1] "x" can take values like "brake", "drive", "throttle", and "drive"

we have codified this in the form of an 'entryMode' argument of the enter function, and different segments of the enter steps are condition with the mode argument. The 'exec' function is responsible for performing a state machine step. Per the Mathworks Stateflow semantics 4 steps are required. These are: a) execute outer flow graphs i.e. outgoing transitions from a state, and if one of the transition is enabled and gets taken then there is no further execution in this state, b) perform 'during' actions, c) execute inner flow graphs, which are transitions from the edge of a state and leading inwards i.e. destination state is a descendant, and if this does not cause any state transition, then d) execute active children in case of a sequential (OR-decomposed) state, and all children in a specific order in case of a concurrent (AND-decomposed) state. The generated code contains a call to the exit function and the enter function respectively for implementing these steps. The exit function is responsible for exiting a state and performs 4 steps unconditionally, which involves exiting sibling in case the state is a parallel state, exiting active children, performing user-defined exit actions, and finally marking the state inactive.

The overall code generation algorithm is in effect a multi-pass traversal over the input graph. There are seven main traversal passes in the transformation. Page limitations prevent us from giving a detailed explanation of the algorithm. In brief, the transformation involves creating `Enter`, `Exit`, and `Exec Function` objects and populating these functions by constructing additional objects (instances of `Statement` and `Statement` derived classes) that faithfully implement the state machine behavior as explained above. For example, the Stateflow defined priority semantics for transitions is enforced by adhering to the following rules: (1) local transitions have a lower priority than cross-hierarchy transitions, (2) the clockwise orientation of transitions determines the priority of outbound transitions. The Stateflow priority semantics is used by domain experts (who are not necessarily software developers) to express domain knowledge. They can assign priorities to transitions in terms of Stateflow concepts within a GReAT transformation, and then the executable transformation is generated automatically from these high-level specifications.

The code generator as implemented offers fairly exhaustive coverage of the Stateflow. The unsupported language features include History junctions, Event broadcast, and arbitrary Junction structures requiring backtracking. The code that gets generated from our code generator is efficient and it preserves the state structures through the state labels. We have systematically validated the functional correctness of the code generator with several examples ranging from small to moderate complexity. We should note here that our goal was not to outperform the code generated by commercial products, which have been developed with an effort of several man years, and refined over an extended period of time.

The transformation includes 82 basic rules, and 54 compound rules. Additionally, there are 54 reuse instances of these basic and compound rules. This compares very favorably with a code generator of similar scale and complexity implemented with ~3000 lines of C++ source code. Using the GReAT tool suite, the code generator can be executed interpretively, or transformed into code, and compiled. For a typical 20-30 states/sub-states model, the interpreted version takes nearly 4 minutes, while the compiled version takes a little over 6 seconds.

4. Conclusion

In this paper we have highlighted a non-trivial model transformation tool: a model-to-code generator whose specification was precisely captured in a modeling language for model transformations, and its implementation was almost entirely automatically generated using a tool that implements the semantics of the modeling language in a translational style. We believe this effort was a successful demonstration of the feasibility and effectiveness of applying MDSD practices to express and automate the implementation of a non-trivial tool. The generator built using the approach works with an acceptable performance, but it also lends itself to formal verification.

We strongly believe that the GReAT and similar technologies offer a unique opportunity for building verifiable generator and transformation tools. However, much future research is needed to build up such a verification technology.

5. References

[1] Karsai G., Agrawal A., Shi F., Sprinkle J., "On the Use of Graph Transformations for the Formal Specification of Model Interpreters", JUCS, November 2003.

[2] Neema S., Karsai G., "Embedded Control Systems Language for Distributed Processing," ISIS Technical Report, ISIS-04-505, May 12, 2004.

[3] Mathworks Stateflow semantics documentation, http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/semantics.html#1032458