Model Transformations in the Model-Based Development of Real-time Systems

Tivadar Szemethy, Gabor Karsai, Daniel Balasubramanian
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA
{tiv,gabor,daniel}@isis.vanderbilt.edu

Abstract

In this paper we argue for UML-based metamodeling and pattern-based graph transformation techniques in computer-based systems development through an illustrative example from the domain of embedded systems. We present a tool that uses advanced graph-rewriting techniques to generate a schedule that satisfies hard real-time constraints for multi-modal systems. The input is a time-triggered system specification (using the Giotto language); the output is an instruction sequence for the E-machine: a virtual machine for hard real-time embedded systems. The resulting model may be refined into a) system implementations (E-code programs) through a trivial synthesis process and b) development-time analysis models expressing the properties of the system implemented over different execution platforms. Furthermore, we identify the next steps to be taken towards generating analysis models using explicit platform models.

1. Introduction

Model-driven development supports the synthesis, the construction, and the design-time analysis of computer-based systems (CBS). However, modeling languages used to capture designs are often far removed from the languages of the actual implementation or analysis. Here, by "language" we mean not only a programming language, but also the constructs provided by an execution platform (e.g. a real-time operating system) one has to use in the implementation phase. Note that for design-time analysis we often use specialized tools (e.g. model checkers) that use yet another language based on a sound mathematical framework.

There is typically a conceptual gap between the concepts available in design languages and those available in platforms and/or analysis tools. Designers like working with domain-specific abstractions [9], design patterns [2], and aspects [8], and yet considerable knowledge and skill is required to map these into the conceptual resources provided by an implementation or analysis platform provided by a traditional computer platform. This is especially relevant for embedded system development, where safety and economic concerns mandate assurances that the design will work as expected in a physical environment.

In short, there is a significant need for relating design models to implementation and analysis models, in order to fulfill the vision of model-driven development. This need can be answered with the help of model transformation techniques, and in this paper we give an illustrative example —from the domain of embedded systems— to show how it can be done. After brief description of the modeling and transformation framework being used, we introduce the example that shows how to extend an earlier, simpler translator (discussed in [13]) with support for multi-modal systems. Finally we summarize the results and examine the further steps necessary for generating platform-specific analysis models using explicit platform models.

2. Modeling and model transformation framework

The Generic Modeling Environment (GME [7]) is a meta-programmable toolkit based on a sophisticated visual model editor. The meta-model (or paradigm in GME) for each domain-specific modeling language (DSML) is defined using the UML-based MetaGME

modeling language. Then, the GME toolkit is configured for the manipulation of paradigm-specific DSMLs.

Apart from the visual model editor, GME provides an API to access the models by paradigm-specific programs (called model interpreters). This API exposes the internal representation of the models, which is a network of object instances and links (associations). A typical interpreter explores this network in a graph-traversal fashion and extracts the information it needs or manipulates the model. Each node (model object) and link (association) has a specific type defined in the metamodel, and attributes can also be associated with them.

Defining a DSML (meta-modeling) in GME consists of the following steps:

(1) Defining the structure and elements of the models in UML using MetaGME.
(2) Specifying visualization rules (for the model editor) by defining model aspects and visual representations (icons).
(3) Specifying well-formedness and consistency rules for the DSML via OCL constraints attached to the metamodel.

The GME toolkit incorporates the Graph Rewriting and Transformation Language (GReAT) [6][7] as a way of visually specifying model interpreters. GReAT uses pattern-based graph rewriting to manipulate (*match, create, modify, delete*) objects of the target model.

Rules are the basic production units, specifying graph patterns in terms of the source and target metamodels. Rules are explicitly sequenced.

Rule blocks provide the means to organize rules into higher-level hierarchies. Within a rule block, rules are chained (and thus sequenced) by passing previously matched elements from rule to rule. Using rule block constructs, a complex transformation can be decomposed into a sequence of simpler rules. GReAT also provides non-deterministic rule execution.

Block/ForBlock units provide the primary means for grouping rules and organizing rule blocks into hierarchy.

Test/Case constructs provide conditional rule invocation similar to "if-else" statements in textual programming languages.

References can be used to implement recursion and code re-use. Rule inheritance is also supported.

Guards and **AttributeMappings** are small procedural code segments that can also be included in rules; either for manipulating object attribute values by using the GME model API, or as guard conditions in matches.

Pattern elements in rules refer to source and target metamodel objects, implying typing. Specifically, pattern objects of a given metamodel type will match instances of the same type in the models being transformed, and if a base type is specified in the pattern, instances of its derived types will also match.

GReAT transformations can also specify objects and associations not explicitly present in the input or output metamodels, including cross-metamodel associations. These entities are called *CrossLinks* and their instances exist only while the transformation is being performed.

Defining a GReAT transformation consists of the following steps:

(1) Importing the source and target GME metamodels.
(2) Specifying the graph rewriting rules using the imported metamodel objects.
(3) Defining (implicit and explicit) sequencing for the rules by grouping them into rule blocks.
(4) Configuring the transformation by specifying source and target models (files) and specify the starting rule (or rule block).

One important advantage of using GReAT is that it allows us to specify our model transformations using the same framework in which our models were created, so that a single framework is used throughout the entire development process.

3. Example: a Giotto → E-code translator

An important area of embedded (control) systems is the area of *(hard) real-time systems*, where the system has to respond to external stimuli before a *deadline*. *Hard* real-time systems are the ones where missing the deadline is considered to be a catastrophic failure of the system (e.g. safety-critical systems). Such systems are often designed according to the t*ime-triggered paradigm* [11] to ensure (timing) correctness by construction. In a time-triggered system, each activity is invoked exactly at a predefined point in time.

Many embedded control systems can be defined as a (periodically repeating) sequence of time-triggered activities (e.g. read sensor; compute control response; drive actuator). These systems are often complex, involving many interdependent sensors, actuators and control laws to be run at different frequencies according to the physical environment the system is designed for.

An example for a high-level language to describe such systems is the *Giotto* language [3] [4], featuring concepts such as sensors, actuators and control computations associated with read/update frequencies in real time. Such languages are implemented over lower-level execution platforms, which provide concepts and primitives closer to traditional CBS (e.g. functions, tasks, device drivers, memory locations etc.).

In [13] we showed the feasibility of a DSML → Platform transformation in GME / GReAT, namely how to translate the high-level model of a time-triggered system (written in *Giotto*), into *E-code*: an abstract code for a virtual machine [5] that schedules tasks in such a way that timing constraints are always satisfied. This schedule is *platform-independent* in the Giotto sense: the externally provided functionality implementations are assumed to be "well-behaving", i.e. they do not violate their deadlines, provided that the scheduler releases them at the beginning of their time slot (which the E-machine can guarantee).

The E-code is similar to the schedules typically used in cyclic executives [10]. Below, we describe how to extend our previous transformation in order to support *multi-modal* Giotto systems. We begin by giving a very brief introduction to the Giotto and E-code languages (concentrating on the issues relevant to multi-modal systems), followed by the description of the transformation extension. A more detailed introduction (relevant to this context) to Giotto and E-code can be found in [13], showing the UML metamodels developed and discussing the single-modal transformation.

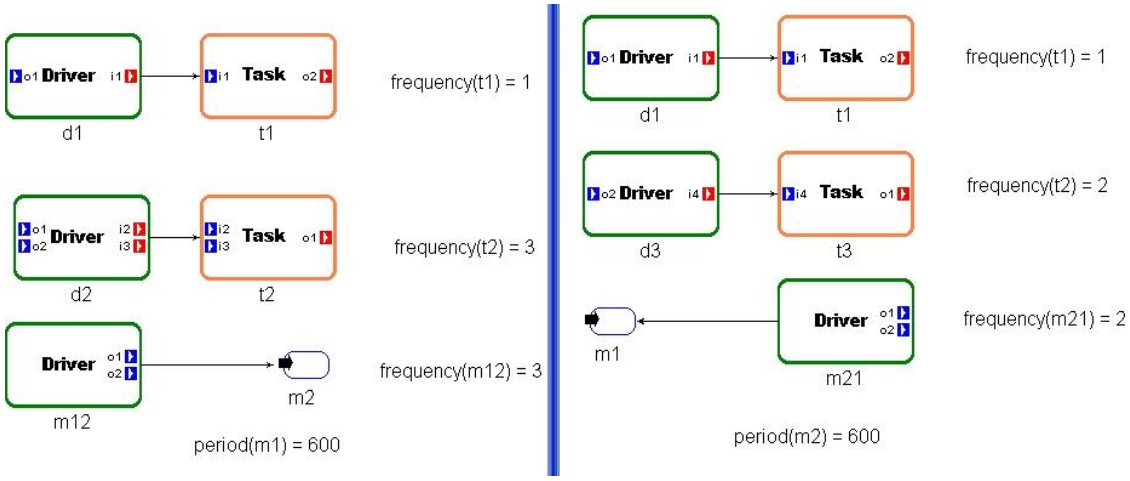

Figure 1 Simple multi-modal Giotto system in GME

3.1 The Giotto time-triggered language

Giotto is a hard real-time embedded systems design language. It is designed for control applications requiring periodic *activities* such as sensor readings, control law computations and actuator updates. According to the time-triggered paradigm, each activity of the system must be strictly periodic. Giotto allows multiple operating *modes* for the system: in each mode, a different (possibly overlapping) subset of activities is executed periodically. Different modes might have different *periods* and each activity is run with a *frequency* relative to the mode period.

In the above case study, a GME metamodel for Giotto systems was created, and the main concepts of the language modeled:

Tasks are the basic functional entities, implemented by external (Java or C++) code. Tasks are expected to run periodically, with a fixed period per mode.

Ports are memory locations (typed variables) facilitating inter-task communication and carrying system state. *Sensors* and *actuators* are also represented as ports.

Drivers perform data copying between ports and implement device access (for sensors and actuators). Drivers execute in *zero logical time*, i.e. the environment does not change while a driver runs. Drivers can also have an associated guard condition, which can be evaluated in *zero logical time* as well.

Modes group periodic *task invocations* and *actuator updates* along with their associated *driver*

calls. The system can transition between modes if a *mode switch driver* guard conditions evaluates true.

Figure 1 shows a multi-modal Giotto system: mode m_1 (shown on the left, with a period of 600ms) runs task invocations t_1 and t_2 (at $\omega_1 = 1$ and $\omega_2 = 3$) and evaluates mode switch guard $m_{1\to2}$ at $\omega_{1\to2} = 3$. Mode m_2 (shown on the right, with the same period of 600ms) runs task invocations t_1 and t_3 (same frequency for t_1 and $\omega_3 = 2$) and evaluates mode switch guard $m_{2\to1}$ at $\omega_{2\to1} = 2$.

Observe that both mode switch guards ($m_{2\to1}$ and $m_{1\to2}$) run at higher frequencies than t_1 (in both modes). This means that the system might switch modes while t_1 is *logically running*. The mode switch in this case is said to *logically interrupt* the execution of t_1. In such cases, the compiler has to make sure that the end of any logically interrupted task's period coincides with the end of its periods in the destination mode. In a well-formed Giotto system, logically interrupted tasks must be contained by both the source and destination modes, and their respective frequencies should be *compatible* (i.e. the above rule should be satisfied). This rule is explained in detail in Part II. Section "Mode switches" in [4].

3.2 The E-Machine and its language the E-code

The E-Machine virtual machine governs the interactions between the software tasks; the real-time OS running the system and the environment. The concepts of Giotto and the E-Machine are related, with the E-Machine having a more general scope (not restricted to periodicity and time-triggered behavior). However, the E-Machine guarantees predictable timing and behavior.

The main functional concepts of the E-Machine have very similar Giotto counterparts (*Task*, *Drivers* and *Ports*). Drivers are separated into two distinct entities, a *Guard* and an (E-Machine) *Driver*.

The E-code language defines the following instruction set:

`if` conditional branching according to the associated *guard*

`call` execute a *driver*, in zero logical time (the E-Machine blocks until the call finishes)

`schedule` place a task instance into the *Ready* queue of the host OS (annotated by its deadline). It is the responsibility of the OS to execute the task on time. The E-Machine resumes execution immediately

`jump` an unconditional branching instruction

`future` "delayed `jump`": the E-Machine yields control to the host OS until the associated *timer guard* evaluates true, and then transfers control to the designated E-code instruction

In GME models, arrows (directed connections) indicate instruction sequencing (`Next`) and dashed arrows show a `Then` branch (after an `if`).

3.3 Extending the Giotto→E-code translator

As mentioned in the Introduction, mapping from the high-level system models to an executable at the implementation platform level is not a trivial task. This is well illustrated by our particular example: a Giotto system is essentially defined as a set of periodic timing constraints, and the E-machine accepts an imperative program (instruction sequence). The translator generates a schedule (composed of E-code instructions) that satisfies the timing constraints.

3.3.1 Concepts of the single-mode translator

In [13], questions regarding establishing the modeling framework, metamodeling Giotto and E-code were discussed. Additionally, a limited prototype Giotto→E-code translator was specified. In summary, the following contributions were made:

(1) GME Metamodels were created for the input (Giotto) and output (E-code) languages,
(2) A GReAT transformation was built to map the Giotto models into sequenced E-code instructions for single-mode Giotto systems.

The main problem for this transformation was to map a single mode onto a repeating E-code instruction chain. The basic idea is as follows:

Determine when the task/drivers need to be executed based on their frequencies, and generate synchronous (zero logical time) instructions (`if`, `call`, `schedule`) to evaluate/execute the required actions, then suspend the E-machine using a `future` until the next action needs to be taken. During this time the host OS executes the scheduled tasks.

The smallest duration to be spent between these zero-logical-time instruction chains is characteristic to the mode (as discussed in [13]), and is called "*unit size*" or "time slot length".

In [13], this interval was appropriate for the whole system, since the transformation supported single-mode systems only. Using this result, the translator generated the appropriate synchronous

`if`$^{\text{then}}$→`call`→`schedule` E-code sequences implementing the system activities. These synchronous (zero logical time) sequences were "separated" in time by delays of "*unit size*" duration implemented by `future` instructions.

The concept can be seen in Figure 2 (the model contains additional code to implement multiple modes, explained below).

In order to extend the above translator for multi-modal systems, the following steps need to be taken:

(1) this interval has to be calculated for each mode, and separate E-code blocks implementing each mode have to be generated
(2) Giotto *mode switches* have to be implemented as conditional jumps between these code blocks to facilitate mode changes

The first task can be trivially accomplished by re-using the previous transformation. Additionally, a `jump` instruction needs to be generated as the first instruction of the program (after initialization) pointing at the code block implementing the initial mode.

The second task is slightly more difficult. *Mode switches* in Giotto are specified as periodic activities within a mode: the *mode switch guard* needs to be evaluated with the specified frequency, and when it evaluates true, control needs to be transferred to the code block implementing the appropriate mode.

As we have seen in the previous section, Giotto allows tasks to be *logically interrupted* by mode switches. If the task is present in both the current and next modes, its time slot has to be aligned with both modes' periods (i.e. it has to "logically finish" at the right time in the next mode even if it was started in the previous one).

This has two consequences:

a) the destination (within the next mode) of the `jump` implementing the mode change has to be calculated accordingly (i.e. it is not always the first instruction of the mode's code block) and

b) the actual `jump` might have to be delayed in order to synchronize the two modes.

Figure 2 (with some details omitted) illustrates this by showing the generated E-code for the Giotto system from Figure 1. Marked arrows indicate (possible) mode switches. At Δ_0 = 0ms (both modes) a mode switch is implemented by a `jump`. At Δ_1 = 200ms in m_1 for example, the effective mode switch has to be delayed by 100ms because the E-code instruction getting control is intended to run 300ms after t_1 was scheduled (and 300ms before it is going to be scheduled again in m_2). The mode switch has to accommodate the logically interrupted task instance during $m_1 \rightarrow m_2$. This is implemented using a `future` instruction.

Note that the E-code implementation splits m_1 into three 200ms "time slots" and m_2 into two 300ms slots. 200 and 300 are referred as "unit sizes" for m_1 and m_2 respectively.

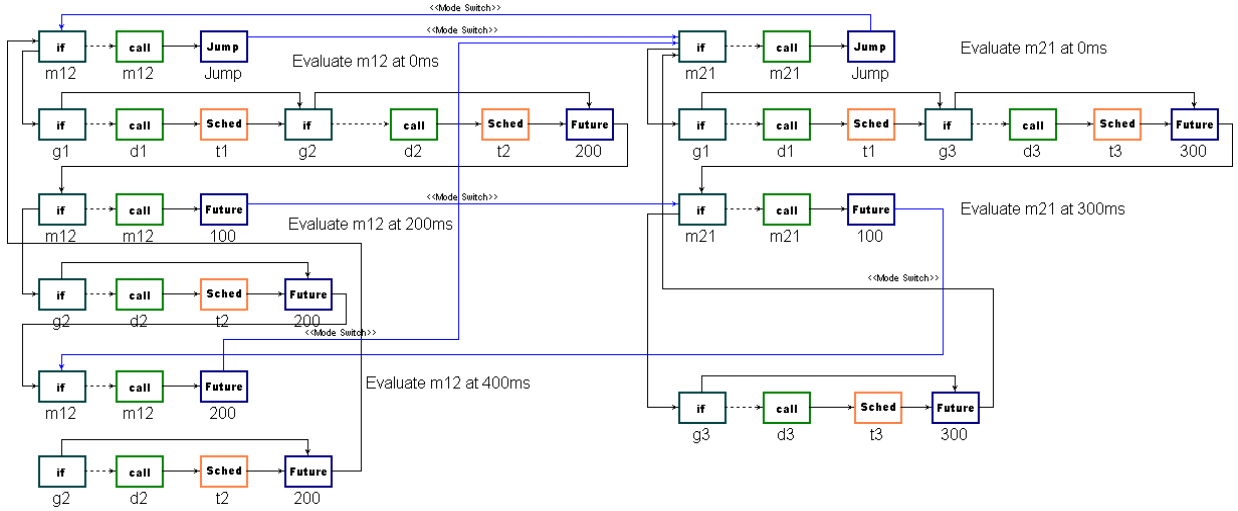

Figure 2 GME Model for E-Code implementing Giotto system from Fig. 1 with mode switches shown

3.4 Implementing the extended translator

The rule block (`Giotto2ECode`) responsible for generating E-code has been extended as shown in Figure 3:

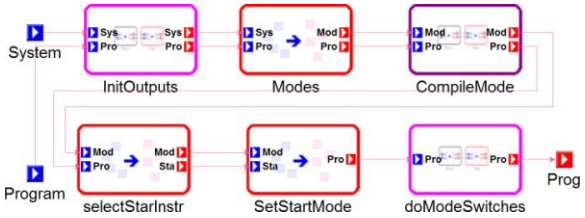

Figure 3 Rule block Giotto2ECode of the extended translator

This chain matches top-level "System" (from the Giotto source model) and "Program" (target E-code model) objects, and routes them through the rule chain shown.

Rule `Modes` (Figure 4) matches all modes in the system, which are then fed into rule block `CompileMode`.

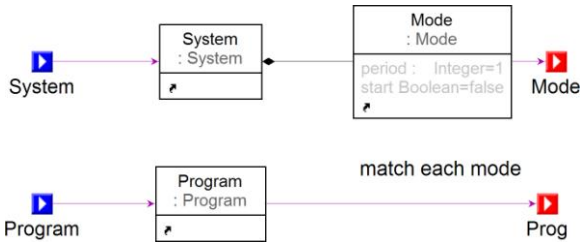

Figure 4 Rule to match all modes in the system

The rule matches inputs `System` and `Program`, and matches all Mode instances contained by `System`. The results (the `Program` instance and all the Mode instances) are propagated to the next rule.

3.4.1 Setting the program entry point

After the E-code blocks have been generated for each mode (rule block CompileMode, discussed below), the initial mode needs to be matched (Figure 5):

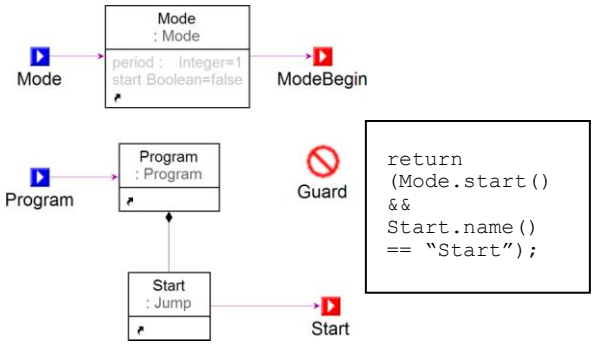

Figure 5 Rule selectStartInstr with Guard code

The C++ Guard condition (using the GME API) selects the `jump` instruction named `Start` from `Program`, and the Mode whose `start` attribute is *true*. This is also necessary since rule block Modes (Fig. 4) matched and propagated all Mode instances.

Then, the initial `jump` instruction is set to the first instruction of the corresponding E-code block, as shown in the next figure. The destination of the `jump` is set to the first instruction of the Mode by creating (indicated by a checkmark) a sequencing (`Next`) connection. (The very first instruction of a Mode is marked with a ModeBegin *crosslink* association.)

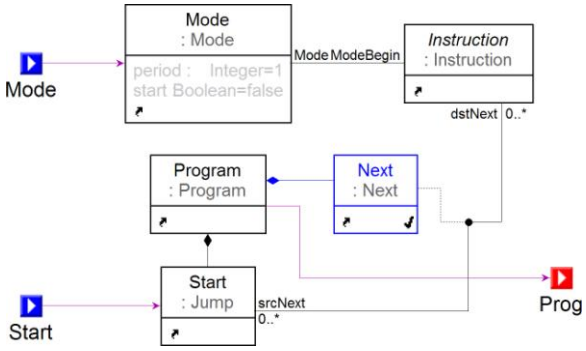

Figure 6 Rule SetStart

3.4.2 Implementing mode switches

As mentioned above, a mode switch is a guarded periodic activity, thus it can be implemented similarly to task invocations. In the Giotto metamodel a ModeSwitch is also derived from the abstract *Periodic*, thus it is matched by rule `PeriodicActions` in block `CompileMode/AddUnits`. In order to process it, the subsequent rule block `AddUnitContent` needs to be extended with a rule explicitly matching ModeSwicthes.

This rule block (`ModeSwitch`) is very similar to existing block `TaskWithDriver`, but rather than generating `if`^{then}→`call`→`schedule` sequences, it creates `if`^{then}→`call`→`jump` sequences.

At this point, the `jump` is left unspecified (without destination), because the code generation has to finish for all modes before the destination point can be determined.

In order to make further processing easier, a *crosslink* (temporary association) is placed between model elements representing the `jump` and the target mode. Furthermore, the time elapsed since the beginning of the period is encoded with the `jump` instruction.

This value is stored in the (otherwise unused) GME attribute *name*. This duration is equal to the sum of the `future` delays within the mode so far, since all other E-code instructions execute in zero logical time. This information is encoded in the name of the last Future instruction, which is also matched.

Rule `TargetMode`, implementing the above is shown in the next figure:

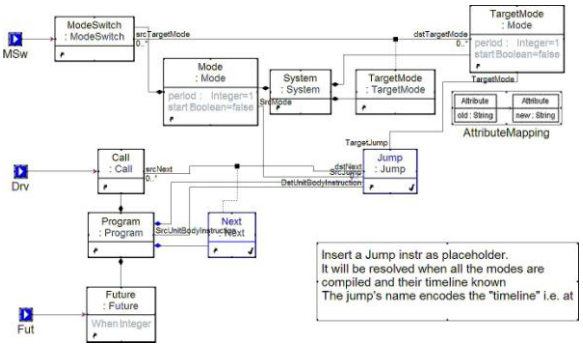

Figure 7 Rule TargetMode

This rule works as follows: *ModeSwitch* and its *TargetMode* are matched, and the `jump` instruction is created (and sequenced in after the driver `call` by creating a `Next` connection). A checkmark in the right bottom corner indicates objects created by the rule, and the "incoming arrow" icons on the left indicate objects matched by previous rules. *TargetMode/TargetJump* shows the *crosslink* connection. The AttributeMapping C++ code block takes care of extracting and encoding (time) values from object attributes.

3.4.3 Determining mode switch target instructions

After the *CompileMode* rule-block has finished (it matches matching all modes), in the final *doModeSwitches* rule-block all `jump` instructions (within the main Program) with a *crosslink* to a Mode are matched (i.e. all "unfinished" modeswitch sequences identified):

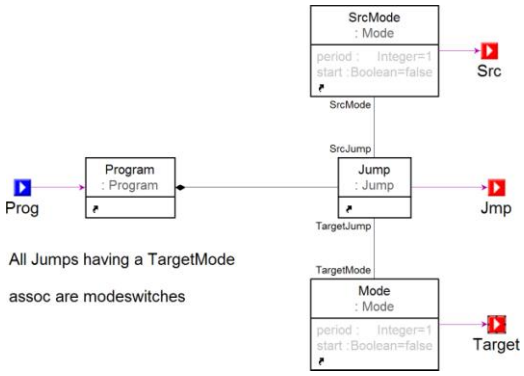

Figure 8 Finding "unfinished" mode switches

The schema for processing the modeswitches is shown below:

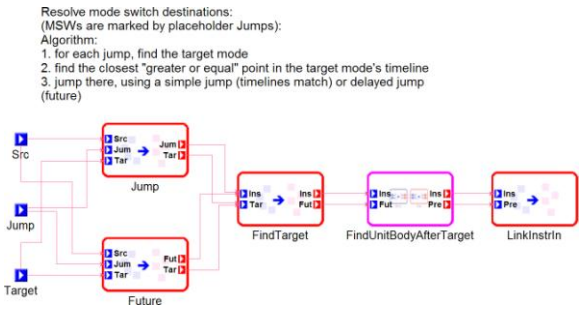

Figure 9 Schema for resolving mode switch destinations

The condition for choosing between a `jump` or `future` instruction can be formulated as follows:

Mode switch $m_{s\to d}$, evaluated at time step Δ in s (modulo p_s, period of mode s) is implemented by a `jump` if the "unit size" of mode d (u_d) is a divisor of Δ. (i.e. the time of the mode switch coincides with the start of a "time slot" in d).

Otherwise, a `future` has to be used and the mode switch has to be delayed until the start of the next time slice in mode d. (Consider Figure 2: All but the very first mode switches at $\Delta = 0$ in both modes are delayed).

This decision is implemented in rules `Jump` and `Future` in Figure 9 by C++ guard conditions extracting the timing information from the model elements and comparing them. Rule `Future` also

takes care of replacing the `jump` instruction by a `future`:

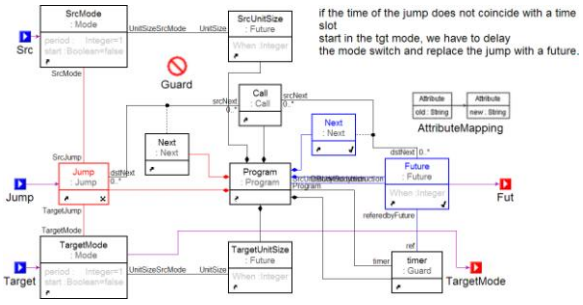

Figure 10 Rule Future in doModeSwitch

The `Guard` condition contains the C++ code for the modulo arithmetic, and if it evaluates true, the `jump` instruction is deleted (small x in the right bottom) and a `future` is created (denoted by a checkmark) and linked into the model by creating all the necessary associations. The temporary `SrcUnitSize` and `TargetUnitSize` elements (on the top and bottom), associated with the source and target models are used to store the unit size values (in their `When` integer attribute). They are temporary variables, created by the `CompileMode` rule block during the processing of the systems' modes.

Then, in Rules `findTarget` / `findUnitBodyAfterTarget` the exact instruction to jump to is determined by walking through the instructions of the target mode until the first instruction of the desired time slot is found, and rule `LinkInstrIn` connects the branching instruction to the appropriate E-code instruction in the target mode.

Extending the transformation required adding or modifying approximately 20-25 rules and took less than two weeks including preparing test cases and verifying the results against the Giotto compiler from UC Berkeley. An e-mail exchange with the Berkeley team was also necessary to clarify a few minor details.

4 Related research

With visual, model-based design languages becoming more and more mainstream, model transformation is getting more and more research attention. Formal model transformation approaches, such the declarative, graph-transformation based GReAT offer a higher level of abstraction and promise better accountability for the transformation process itself.

Pioneering efforts in this area are lead by OMG with the QVT standardization efforts [14][15]. The OMG advocates declarative and hybrid model transformation approaches with the OCL language (with extensions) having a key role in the proposed standard. GReAT uses C++ code for attribute manipulation and match guard predicates, because of the availability of existing C++ model manipulation APIs. Also, OMG uses the MOF language as opposed to UML, but as it was shown in [16] the difference in the context of model transformation is not really relevant. Nevertheless, important research is being done with native MOF and OCL compliant model transformation techniques by the ATL [17] group.

Model transformation also enables creating formal *analysis models* derived from high-level design models. Analysis models capture the system's behavior in various abstract formalisms (such as timed automata [1]) which enable formal verification through *model checking*. The VIATRA [18] framework, relying on the graph transformation of UML models into formal analysis models is a good and innovative example of such approaches.

It is quite often a requirement to analyze a design's behavior implemented over different platforms (offering the same interface). The (execution) platform is a natural choice for abstraction (and standardization) – this abstraction is probably best described by the Metropolis approach [19].

5 Future work

Using the platform-level abstraction, in a recent paper [13] we argued for the modeling of the execution platform that facilitates the component interactions in component-based embedded systems. Using *platform modeling*, we showed an example for deriving formal analysis models through a model transformation incorporating information on the implementation platform.

In the example, we constructed a transformation that took a domain-specific system-design model and translated it into timed automata that modeled the behavior of the design implemented on a specific execution platform.

One important lesson we learned from the experiment was that the transformation contained a section implementing the DSML→Platform mapping. This part translated the DSM onto platform-level structures, and then these structures were further

transformed onto analysis model structures describing their behavior.

We realized that it is worth formally separating this transformation, since it could to be re-used if we want to construct additional DSML→Analysis model transformations either

a) describing a different implementation of the same platform, or
b) generating an analysis model in a different analysis formalism, describing the same system over the same platform

This was the main motivation behind the work described in this paper.

An additional (very important) benefit of having a DSML→Platform mapping is that it can be used for system synthesis, as it is demonstrated by the example discussed in this paper. Using the resulting E-code models, an actual E-code program implementing the system could be generated trivially.

Of course, tools (compilers) already exist to perform this task, such as the Giotto E-code compiler from Berkeley.

Having the very same transformation generate both the (model of the) implementation code and the analysis model guarantees that the two models (implementation and analysis) are isomorphic and the analysis results are valid. This is the main benefit gained by the (re)implementation of the compiler in the above model transformation framework.

Next, we are going to work on formalizing (and making it explicit) the *platform model*: the part of the DSML→Analysis model transformation describing the platform-level entities (in terms of analysis model structures) and their interactions.

6 Conclusions

This paper focused on the first step towards the solution of the *explicit platform modeling* problem discussed above:

We demonstrated an example for the DSM → Platform model transformation and showed how it fits into the generic modeling and transformation framework. For us, this transformation is important because it provides a system model at the *platform level*, thus enables incorporating platform implementation knowledge into further model translations towards analysis models.

Our work also underlines a very important benefit of the model-driven development (MDD) approach: The very same transformation can be used in system synthesis and analysis model generation, thus reducing development effort and – more importantly – ensuring the validity of the analysis models for the actual implementation.

Model-driven engineering techniques are a prime source for innovation and management of complexity in embedded system development. In this paper we have given an illustrative example that shows how model transformation (specified in a high-level, formal language that uses graph rewriting techniques) can be used to solve non-trivial mapping problems between domain-specific models and platforms. The example is meant to emphasize the opportunity a model transformation system can provide, and are not necessarily optimized for performance.

We believe the next major research problems in this area include (1) developing the technology for the more general, platform-model-driven transformations for analysis, and (2) developing the technology for correctness-preserving transformations. Both of these require further, significant research efforts.

7 References

[1] R. Alur and D. Dill: “Automata for modeling Real-Time Systems”. Theoretical Computer Science, 126(2): 183–236, April 1994

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Addison-Wesley, 1995.

[3] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch: “Embedded control systems development with Giotto”, Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM Press, 2001, pp. 64-72.

[4] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch: ”Giotto: A time-triggered language for embedded programming.” Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, Springer-Verlag, 2001, pp. 166-184.

[5] Thomas A. Henzinger and Christoph M. Kirsch: ”The Embedded Machine: Predictable, portable real-time code” Proceedings of the International Conference on Programming Language Design and Implementation (PLDI), ACM Press, 2002, pp. 315 326.

[6] Karsai, G., Agarwal, A., Shi, F., Sprinkle, J: “On the Use of Graph Transformation in the Formal Specification of Model Interpreters”, Journal of Universal Computer Science, Volume 9, Issue 11, 2003.

[7] Agrawal, G. Karsai, F. Shi: “Graph Transformations on Domain-Specific Models”, Technical Report, available online at http://www.isis.vanderbilt.edu

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin: ”Aspect-oriented programming.” In ECOOP'97---Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220--242, 1997.

[9] Ledeczi, A.; Bakay, A.; Maroti, M.; Volgyesi, P.; Nordstrom, G.; Sprinkle, J.; Karsai, G.: ”Composing domain-specific design environments”, IEEE Computer, Nov. 2001, Page(s): 44 –51.

[10] A. Shaw: ”Real-Time Systems and Software”, John Wiley & Sons Inc., 2001.

[11] H. Kopetz: “The Time-Triggered Model of Computation”, Proceedings of the 19th IEEE Systems Symposium (RTSS98), December 1998

[12] T. Szemethy and G. Karsai: “Platform Modeling and Model Transformations for Analysis”, Journal of Universal Computer Science, Volume 10, Issue 10, 2004 pp 1383 1407

[13] T. Szemethy: “Case Study: Model Transformations for Time-Triggered Systems”, International Workshop on Graph and Model Transformations (GRaMoT), Tallinn, Estonia, September 2005.

[14] Gardner et al. “A review of OMG MOF 2.0 Query / View / Transformation Submissions and Recommendations towards the final Standard” available online at http://www.zurich.ibm.com/pdf/ebizz/gardner-etal.pdf

[15] The QVT-Merge Group: “Revised submissions for MOF 2.0 Query/Views/Transformation RFP”, available from the QVT Group

[16] M. Emerson, J. Sztipanovits, T. Bapty: “A MOF-Based Metamodeling Environment”, Journal of Universal Computer Science, Volume 10, Issue 10, 2004 pp. 1357-1382

[17] J. Bézivin et al: “First experiments with the ATL model transformation language: Transforming XSLT into Query” In the online proceedings of the OOPSLA’03 Workshop on Generative Techniques in the Context of the MDA, http://www.softmetaware.com/oopsla2003/mda-workshop.html

[18] Gy. Csertán, G. Huszerl, I. Majzik, Zs. Pap, A.Pataricza, and D. Varró: ”VIATRA: Visual automated transformations for formal verification and validation of UML models.” In J. Richardson, W. Emmerich and D. Wile Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering, pages 267-270, Edinburgh, UK, September 23-27 2002.

[19] F. Balarin et al.: “Modeling and Designing Heterogeneous Systems” Concurrency and Hardware Design, Springer 2002, pp 228-273