Design-time Simulation of Domain-Specific Models by Interactive Model Transformations

István RÁTH
Advisor: Dániel VARRÓ

I. Introduction

In industrial applications (such as embedded system design), domain-specific modeling languages (DSML) are frequently used to model dynamic systems. Futhermore, models are not only used to generate the target application's source code, but also for performing design-time simulation, analysis, validation, and verification using special tools.

Thus, an important issue that needs to be addressed when developing new DSMLs is the precise specification of the semantics of the language. Since this is mostly lacking in current language engineering tools [1, 2], these features have to be implemented using manual coding, which is time consuming and expensive.

In the current paper, we present the new facilities of the ViatraDSM domain-specific modeling framework [3], which aims at bridging the gap by an integrated support for (i) model simulation and (ii) model-to-model transformations. ViatraDSM relies on the underlying VIATRA2 model transformation framework [4] to provide a unified high-level formalism for the specification of dynamic behavioral semantics of a language and to capture model-to-model transformations.

II. Model execution in domain-specific languages

Discrete model execution is applicable to domains where the state-space can be evaluated in discrete time intervals, i.e. changes are *atomic*. While this may also include the discrete approximation of complex continuous time systems such as signal networks, in practise, our approach is targeted at those domains where all concepts can be captured using a bounded (small) number of dynamic entites with each having a well-defined life cycle. This includes "token game" simulations and also systems involving dynamically created object instances (birth-death processes).

Model execution can be either fully automatic or user-guided. User assistance is a requirement in many cases, e.g. when design-time "debugging" is performed by designers. Another source of *interactivity* may be non-determinism, which is frequently present in practical model simulation scenarios. In that case, the user is given a set of choices at a *choice point*, and the execution continues depending on the actual choice of the user.

Therefore, we have chosen to build domain-specific interactive model simulators by using (i) a statemachine formalism to drive user interaction on a very high level of abstraction (Sec. IV.), and (ii) the model transformation language of VIATRA2 to capture atomic simulation steps (Sec. III.), which preserve domain-specific validity constraints.

III. Domain model execution in VIATRA2

First we present how to precisely capture elementary model simulation steps by combining two formal methods to design model simulation for Petri nets. For this purpose, a brief overview is provided to the transformation language of VIATRA2 based upon [4].

A. *Overview of* VIATRA2 *transformation language*

The Viatra Textual Command Language (VTCL) consists of several constructs that together form an expressive language for developing both model to model transformations and code generators. Graph patterns (GP) define constraints and conditions on models, graph transformation (GT) [5] rules support the definition of elementary model manipulations, while abstract state machine (ASM) [6] rules can be used for the description of control structures.

Graph patterns, negative patterns Graph patterns (Fig. 1) are the atomic units of model transformations. They represent conditions (or constraints) that have to be fulfilled by a part of the model space in order to execute some manipulation steps on the model. Patterns may have parameters listed after the pattern name. The basic pattern body contains model element and relationship definitions defined by VTML constructs ([7]).

```
// Describes a Place and its Token instance.
pattern placeToken(P, T) =
{
// variable P is an instance of 'Place'
'PetriNet'.'Place'(P);
// variable T is an instance of 'Transition'
'PetriNet'.'Transition'(T);
 // they connected by a 'tokens' relation
'PetriNet'.'Place'.'tokens'(R, P, T);
}
```

Figure 1: Graph pattern: a Place and its Token.

A model (i.e. part of the model space) can satisfy a graph pattern, if the pattern can be matched to a subgraph of the model using a generalized *graph pattern matching* technique presented.

In VTCL, *patterns may call other patterns* using the *find* keyword. This feature enables the reuse of existing patterns as a part of a new (more complex) one.

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Place) =
{
 precondition find placeToken(Place, Token)
 postcondition find place(Place)
}
```

Figure 2: GT rule: removing a Token from a Place.

Graph transformation rules Graph transformation (GT) [5] provides a high-level rule and pattern-based manipulation language for graph models. In VTCL, graph transformation rules (Fig. 2) may be specified by using a *precondition* (or left-hand side – LHS) pattern determining the applicability of the rule, and a *postcondition* pattern (or right-hand side – RHS) which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are deleted, elements that are present only in RHS are created, and other model elements remain unchanged. The LHS and RHS patterns share information on matchings by parameter passing.

Complex transformation programs To execute graph transformation rules, they have to be invoked from a transformation program. In this case, the actual parameter list of the transformation has to contain a valid value for all input parameters, and an unbound variable for all output parameters. A rule can be executed for all possible matches by quantifying some of the input parameters using the *forall* construct.

```
rule fireTransition(in Tr) = seq {
  forall P with find sourcePlace(Tr, P) do
      apply removeToken(P);
  forall P with find targetPlace(Tr, P) do
      apply addToken(P);
}
```

Figure 3: MT rule: firing a Transition.

To illustrate how a VIATRA2 transformation program can be used to execute a simulation step, a small VIATRA2 transformation program is shown on Figure 3. It takes a Transition instance as input, locates all Places which are connected to the Transition by an OutArc (sourcePlace pattern), removes a token from each of those Places (removeToken rule), locates all Places which are connected to the Transition by an InArc (targetPlace pattern), and adds a token to each of those Places (addToken rule). In this case, the `forall` ASM construct is used to compute all matches of the sourcePlace and targetPlace patterns, and two simple graph transformation rules are applied to facilitate the removal and addition of Tokens.

Enabledness calculation A graph pattern expressing the enabledness condition for the simple fire simulation step can be used with the `forall` construct, which generates all matchings for that pattern.

IV. User-guided transformations for model simulation

To facilitate design-time interactive model execution, the ViatraDSM framework provides domain-specific user-guided simulation built on top of VIATRA2 model transformations. The fundamental idea is that each atomic model simulation step (modifying the instance model but preserving syntactic validity) is a VIATRA2 model transformation, defined over the abstract syntax metamodel of the domain. User interaction may be provided when the execution of step is fully finished. The user may continuously observe the changes of the domain-specific model as the simulation steps are executed, and provide input at non-deterministic choice points. This interaction is specified by a state transition system based on guarded commands.

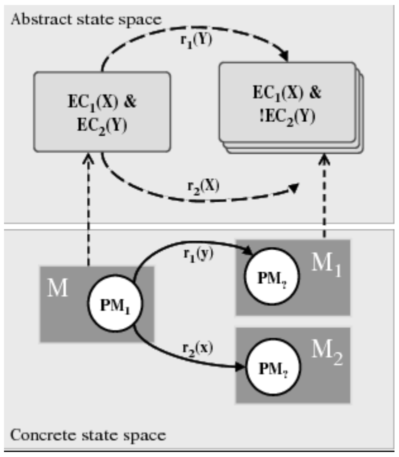

Figure 4: Abstract and concrete state spaces.

A. *The simulation process*

An abstract simulation execution, which is perceived by the user, can be formalised using a state transition system specified by guarded commands. Transformation programs defined in Sec. III. can be grouped into two categories: (i) *enabledness conditions* EC and (ii) *simulation rules* R. Simulation rules can be used to manage control and data flow; it is up to the designer to separate these concerns if the domain necessitates. Enabledness conditions act as guards for the execution of simulation rules. The simulation process is illustrated in Fig. 4 using abstract (top) and concrete (bottom) steps.

The abstract system evolves along transitions by executing a simulation rule r provided that the guard of this rule g_r is satisfied. This abstract guard is a logical formula composed of (positive or negative) enabledness conditions as literals. The abstract transition system captures the high-level view of a simulation as perceived by the designer.

All transitions on the abstract level are *potential* transitions, i.e. there may but not necessarily be a corresponding concrete transition in the system model. For instance, if the ”fire transition” simulation step rule is be fireable for transition T1, but not fireable for transitions T2 and T3, this still becomes a valid transition on the abstract level. However, due to the dynamic nature of the system state, it is frequently infeasible to a priori calculate all the transitions on the abstract level. Instead, the execution of an abstract transition is derived from executing steps on the concrete level as follows:

- First all enabledness conditions (e.g. $EC_1(X)$ and $EC_2(Y)$ in Fig. 4) are evaluated by initiating graph pattern matching on the current concrete state M in order to determine the current abstract state. An enabledness condition is satisfied if there is at least one match on the concrete level. The validity of enabledness conditions determine the abstract state, e.g. if $PM_1(x, y)$ is a part of the system model which satisfies both $EC_1(X)$ and $EC_2(Y)$ along some match $X = x$ and $Y = y$, then one can conclude that the abstract state machine is in the abstract state $EC_1(X) \wedge EC_2(Y)$.
- Then we check on the abstract level, which simulation transitions are enabled, i.e. lead out from the current abstract state (e.g. $r_1(Y)$ and $r_2(X)$ in Fig. 4. Exactly these enabled simulation rules will be offered to the designer for user interaction. If there are multiple matches of a simulation rule on the concrete level, they denote additional user choice points. This way, a user can select the next simulation rule to apply as well as the concrete matches (e.g. $Y = y$) where the corresponding rules are applicable.
- After the user makes his or her choice, the simulation rule is executed for the given match on the concrete level to derive a new model M_1 from M as a result of the atomic simulation step. After this, the process starts over again by evaluating the enabledness conditions in M_1.

B. *Evaluation of enabledness conditions*

While enabledness conditions are most typically expressed using a graph pattern, in some cases it may be necessary to perform more complex computations. Therefore, the ViatraDSM framework uses normal VIATRA2 transformation programs for this purpose as well: ”check machines” traverse the model space and produce appropriate output marking for those elements which may be used as execution inputs for the given simulation step.

Since model space traversal is required at each choice point, our technique is best suited for domains where most simulation steps are limited in scope (*locality* principle) – which is the case with many executable modeling domains. In addition, we are developing a technique to speed up the evaluation of enabledness conditions by *incremental pattern matching* [8]. In this approach, graph patterns only need (partial) re-evaluation after the model has changed, since underlying mechanisms automatically track model space alterations and update the bindings of a previously found match accordingly, which may result in a significant speed up.

V. Conclusions and Future Work

In the current paper, we presented how interactive design-time model simulation can be specified on a high-level of abstraction (i) by the transformation language of VIATRA2 for capturing atomic model simulation steps, and (ii) by a state transition system for modeling user interaction, as integrated in the ViatraDSM framework.

While none of the core techniques (i.e. using model transformations as atomic steps, statemachines to describe user interaction) are unprecedented separately in themselves, our combination is a significant contribution both in DSML frameworks and in the field of model transformation where no support is existent for user-guided model transformations driven at a high-level of abstraction.

In the future, we primarily aim at developing support for incremental model transformations [8], which can significantly speed up model simulation. In this approach, matches are stored explicitly, and refreshed incrementally when the underlying model is changed. As a consequence, we expect a drastic reduction in the evaluation of enabledness conditions.

References

[1] The Eclipse Project, “Graphical Modeling Framework,” http://www.eclipse.org/gmf.

[2] Microsoft, “DSL Tools,” http://lab.msdn.microsoft.com/teamsystem/workshop/dsltools/default.aspx.

[3] I. Ráth and D. Varró, “Challenges for advanced domain-specific modeling frameworks,” Nantes, France, July 2006.

[4] D. Varró and A. Balogh, “The model transformation language of the VIATRA2 framework,” *Science of Computer Programming*, 68(3):214–234, 2007.

[5] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Eds., *Handbook on Graph Grammars and Computing by Graph Transformation*, vol. 2: Applications, Languages and Tools, World Scientific, 1999.

[6] E. Börger and R. Särk, *Abstract State Machines. A method for High-Level System Design and Analysis*, Springer-Verlag, 2003.

[7] D. Varró and A. Pataricza, “VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML,” *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[8] G. Varró, D. Varró, A. Schürr, “Incremental graph pattern matching: Data structures and initial experiments,” in *Proceedings of the 2nd International Workshop on Graph and Model Transformation*, Brighton, United Kingdom, September 2006.