

ENHANCING DESIGN-TIME MODEL EXECUTION IN DOMAIN-SPECIFIC LANGUAGES BY INCREMENTAL PATTERN MATCHING

István RÁTH
Advisor: Dániel VARRÓ

I. Introduction

Visual domain-specific languages (DSLs) are used nowadays in a wide range of applications ranging from embedded to enterprise-scale systems. Many of such languages aim to capture the dynamic, behavioral aspects of the system under design. In order to support early validation of design decisions, these dynamic models are investigated by sophisticated simulation tools.

Simulation based early system validation can be carried out by mapping domain-specific models into existing mathematical simulation tools using model transformations. For instance, (i) “token game” simulators capture the data and control flow in a static network, (ii) “birth and death” simulators enable the creation and deletion of objects, but they are unable to model complex contextual logical conditions for such changes. Moreover, when domain-specific models (DSMs) are projected into existing simulation tools, the result of a simulation is provided on the level of mathematical models, and not directly on the DSM level, which can be a limitation for domain experts. In contrast, we focus on efficient simulation directly within a domain-specific modeling environment based upon the dynamic semantics of the DSL.

Contributions of the paper. In this paper, we present the enhanced variant of our general purpose discrete event simulation framework for domain-specific visual languages describing system behavior. Building on our efficient approach [1], we use *incremental graph pattern matching applied to model execution* as the main conceptual novelty compared to our previous paper [2] and other graph transformation-based simulation approaches. We provide an integrated execution system based on the VIATRA2 graph-transformation formalism, which leverages the incremental pattern matcher for the (i) *efficient tracking of enabledness conditions of simulation rules*, and (ii) *the efficient execution of simulation rules*. As a result, our simulation framework supports in-place model changes (e.g. user-driven editing) as well as transaction-like model changes (e.g. model imports) during simulation; the system dynamically adapts to how the models evolve and updates the enabledness of simulation rules accordingly. Our approach, integrated into the ViatraDSM framework, primarily targets *interactive* applications where the user can experiment with model execution. However, as the performance of the underlying incremental pattern matching engine allows [3], the system can go beyond model animation to perform real *model simulation*. which enables analysis involving a high number of execution runs.

II. Overview of the Approach

A. Discrete model simulation with complex model changes

In this paper, we focus on discrete model simulation which is applicable to domains where the state-space can be evaluated in discrete time intervals. Our approach is targeted at domains where all concepts can be captured using a finite number of dynamic entities, each having a well-defined life cycle (e.g. tokens, stateful model elements). Examples of such domains include various state machine formalisms, token games, or data flow networks. Additionally, our approach supports systems involving arbitrarily complex model changes including the birth and death of objects, reconfiguration of net-

works, etc. Note that dynamic changes are not limited to dedicated model elements of the model (e.g. tokens) but arbitrary model objects may be created and deleted during simulation.

For discrete model execution, our approach makes use of *model transformations* operating on a *graph representation* of model elements. As a demonstrating example, we use Petri nets.

The dynamic semantics is described in our approach by *simulation rules*. A rule $R = (EC, AS)$ is defined by an *enabledness condition* EC and an *action sequence* AS , which describes model manipulation. Formally, an enabledness condition corresponds to a graph pattern, and model manipulation can be described by graph transformation (GT) or abstract state machine (ASM) rules as available in the transformation language of the VIATRA2 framework [4]. A simulation rule is executed in a *model context*, which consists of model elements satisfying constraints prescribed by the enabledness condition (a *match* of the graph pattern).

Enabledness conditions. *Graph patterns* represent conditions (or constraints) that have to be fulfilled by a part of the model. *Patterns may call other patterns* using the *find* keyword, which enables the reuse of existing patterns as a part of a new (more complex) one. A *negative application condition* (NAC, defined by a negative subpattern with the *neg* keyword) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Negative conditions can be embedded into each other (e.g. negations of negations).

Action sequences. *Graph transformation* [5] provides a high-level rule and pattern-based manipulation language for graph models. In VIATRA2, graph transformation rules may be specified by using a *precondition* (or left-hand side – LHS) pattern determining the applicability of the rule, and a *post-condition* pattern (or right-hand side – RHS) which declaratively specifies the result model after rule application.

B. Efficient execution

The efficient execution of the simulation depends on the evaluation of enabledness conditions corresponding to simulation rules. In existing GT-based tools, the re-evaluation of such a condition would require the re-computation of pattern matches, which is expensive. To eliminate the problem, we make use of our novel incremental pattern matcher [1] based on the RETE algorithm. Our approach relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates those model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a *view*. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur. As an example, the pattern matcher RETE network constructed for the *transitionFireable* pattern is shown in Fig. 1.

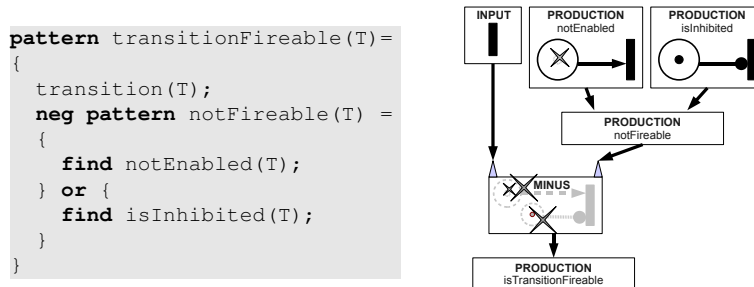


Figure 1: RETE matcher: 'transitionFireable'

Our RETE implementation supports *set operations* at *intermediate nodes*, which can be executed on the match sets stored at input nodes to compute the match set which is stored at the intermediate node (the minus-join node in Fig. 1 represents the negative application condition, while the inter-

mediate production node represents the disjunction by the OR-pattern). After the network has been constructed, the match set for the entire pattern can be retrieved from the output *production node*. Our implementation supports the entire VIATRA2 pattern language, so any VIATRA2 pattern can be matched incrementally.

Incremental updates. The RETE network receives notifications about changes on the model. These changes are propagated through the network, modifying the match sets stored at the nodes incrementally, since each node only recomputes a partial matching. After the changes have been processed, the match set can be retrieved from the network instantly. As a trade-off, there is increased memory consumption, and a moderate overhead on update operations. From the point-of-view of the simulation engine, this means that the *enabledness conditions* can be evaluated at any given time without any expensive pattern matching, for both user-made changes (editing), transformation-induced changes, or even model imports.

C. Integrated design-time simulation

Based upon the set of rules capturing dynamic semantics, our simulation framework supports the execution of a domain-specific model inside the modeling environment. Integrated into the ViatraDSM domain-specific modeling tool, the simulation engine may be invoked at any time; moreover, models are executed at the high abstraction level of the DSL (like in specialised simulation tools), ensuring domain consistency.

Model execution can be either fully automatic or user-guided. User assistance is a requirement in many cases, e.g. when design-time testing is performed by designers. Additionally, interactivity is also used for resolving 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.

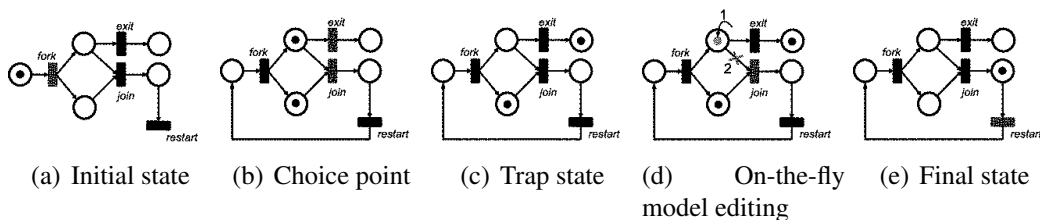


Figure 2: Petri net simulation phases

The most important phases of an example simulation process in the Petri net example are illustrated in Fig. 2. The user can make various changes as the simulation is being executed by *editing the model on-the-fly*. For instance, the user may re-enable the transition *join* by either placing a new *token* into the empty input place (denoted with 1 in Fig. 2(d)), or delete the input arc (denoted with 2). In both cases, the RETE network is automatically updated following the editing action, moving the simulation system back into a state where *join* is enabled for firing. The system also provides dynamic support for the addition, change, or removal of transformation rules (with enabledness conditions and action sequences), since the construction of RETE networks is dynamically performed as pattern definitions are loaded. This feature is analogous to “hot code replace” found in modern program debuggers, and it is very important for agile development.

III. Related work

Industrial DSM frameworks such as Microsoft DSL Tools and GMF both concentrate on a generative approach to ease the development of modeling environments; model execution and transformation in general are yet to be integrated.

Several academic DSM frameworks are complemented with support for model transformations, typically, using a *graph transformation* [5] approach, which approaches show the closest correspondence with our ViatraDSM framework. DiaMeta (a follow-up of DiaGen [6]) replaces hypergraph grammars by MOF as provided by the MOFLON tool suite [7] to allow users not only to specify domain-specific modeling languages but also to generate corresponding diagram editors. Advanced multi-domain modeling features are supported by ATOM3 [8], which defines the concept of view metamodels sharing a common metamodel of a visual language. The consistency of different views and abstract-to-concrete syntax mappings [9] is maintained by triple graph grammars (TGG) while simulators are also defined by graph grammar rules.

IV. Conclusion

In the paper, we presented a discrete event interactive simulation framework for dynamic domain-specific modeling languages. The dynamic semantics of a language was captured by a combination of graph transformation and abstract state machine rules as provided by the transformation language of the VIATRA2 framework.

As the main novelty, our approach is built upon an incremental graph pattern matching engine, which instantly identifies all contexts where the enabledness condition of a simulation rule is enabled. This is carried out by incrementally keeping track of matches of graph patterns related to enabledness conditions. As a result, simulations requiring complex model changes (even with intensive creation and deletion of objects) can be executed in an efficient way.

References

- [1] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, and G. Varró, “Incremental pattern matching in the VIATRA transformation system,” in *GRaMoT’08, 3rd International Workshop on Graph and Model Transformation*. 30th International Conference on Software Engineering, 2008, Accepted.
- [2] I. Ráth and D. Varró, “Design-time Simulation of Domain-specific Models by Interactive Model Transformations,” in *PhD MiniSymposium*. BME MIT, 2008.
- [3] G. Bergmann, A. Horváth, I. Ráth, and D. Varró, “A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation,” in *ICGT’08, 4th International Conference on Graph Transformation*. European Association for Theoretical Computer Science and European Association of Software Science and Technology, 2008, In Press.
- [4] D. Varró and A. Balogh, “The model transformation language of the VIATRA2 framework,” *Science of Computer Programming*, 68(3):214–234, October 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] O. Köth and M. Minas, “Generating diagram editors providing free-hand editing as well as syntax-directed editing,” in *GRATRA 2000 Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems*, H. Ehrig and G. Taentzer, Eds., pp. 32–39, Berlin, Germany, March 25–27 2000.
- [7] M. Minas, “Generating visual editors based on FUJABA/MOFLON and DIAMETA,” Tech. Rep., University Paderborn, 2006, Proc. 4th Fujaba Days, pp. 35–42, Technical Report tr-ri-06-275.
- [8] J. de Lara and H. Vangheluwe, “AToM3: A tool for multi-formalism and meta-modelling,” in *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8–12, 2002, Proceedings*, R.-D. Kutsche and H. Weber, Eds., vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.
- [9] E. Guerra and J. de Lara, “Event-driven grammars: Towards the integration of meta-modelling and graph transformation,” in *Proc. 2nd International Conference On Graph Transformation (ICGT 2004)*, H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, Eds., vol. 3256 of *LNCS*, pp. 54–69. Springer, 2004.