Validating Rule-based Algorithms

László Lengyel

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Magyar tudósok körútja 2, 1117 Budapest, Hungary
lengyel@aut.bme.com

Abstract: A rule-based system is a series of if-then statements that utilizes a set of assertions, to which rules are created on how to act upon those assertions. Rule-based systems often construct the basis of software artifacts which can provide answers to problems in place of human experts. Such systems are also referred as expert systems. Rule-based solutions are also widely applied in artificial intelligence-based systems, and graph rewriting is one of the most frequently applied implementation techniques for their realization. As the necessity for reliable rule-based systems increases, so emerges the field of research regarding verification and validation of graph rewriting-based approaches. Verification and validation indicate determining the accuracy of a model transformation / rule-based system, and ensure that the processing output satisfies specific conditions. This paper introduces the concept of taming the complexity of these verification/validation solutions by starting with the most general case and moving towards more specific solutions. Furthermore, we provide a dynamic (online) method to support the validation of algorithms designed and executed in rule-based systems. The proposed approach is based on a graph rewriting-based solution.

Keywords: verification/validation of rule-based systems; graph rewriting-based model transformations; dynamic verification of model transformations

1 Introduction

Rule-based systems [1] [2] provide an adaptable method, suitable for a number of different problems. Rule-based systems are appropriate for fields, where the problem area can be written in the form of *if-then* rule statements and for which the problem area is not extremely too great. In case of too many rules, the system may become difficult to maintain and can result in decreased performance speeds.

A classic example of a rule-based system is a domain-specific expert system that uses rules to make deductions or narrow down choices. For example, an expert system might help a doctor choose the correct diagnosis based on a dozen symptoms, or select tactical moves when playing a game. Rule-based systems can

be used in natural language processing or to perform lexical analysis to compile or interpret computer programs. Rule-based programming attempts to derive execution instructions from a starting set of data and rules. This is a more indirect method than that employed by an imperative programming language, which lists execution steps sequentially.

As rule-based systems are being applied to many diverse scenarios, there is a need for methods that support the verification and validation (V&V) of the algorithms performed by these systems. In this paper, we discuss the concept of taming the complexity of the verification/validation solutions. Furthermore, we introduce a dynamic (online) approach to address the V&V of rule-based systems.

V&V of a rule-based system is the process of ensuring that the rules meet specifications and fulfill their intended purpose. Based on [3], we use the following definitions: *Verification* is the process of evaluating the rule definitions to determine whether the imposed specification is fulfilled. *Validation* is the process of evaluating the rules, either during or following the rule execution, to determine whether it satisfies the end-user requirements. In other words, validation is intended to answer the question: "Is this the system we intended to create from the users perspective?" (Is this product specified according to the user's actual needs?) Verification provides answers to the question: "Is the system built in accordance with the design?" (Does the product conform to the specifications?)

During the analysis of a rule-based system, our goal is to prove that (i) certain properties hold for the output, if the input is valid, or (ii) to provide the criteria that must be satisfied by the input in order to guarantee the desired properties for the output. The analysis of a rule-based system is said to be *static* when the implementation of the rules and the language definition of the input and output models are used during the analysis process without considering the specific input. In the case of the *dynamic* approach, we analyze the rule-based system for a specific input, and then check whether certain properties hold for the output during or after the successful application of the rules. The static technique is more general and poses more complex challenges. The goal of static analysis is to determine whether the rule-based system itself meets various, specific requirements.

The rest of this paper is organized as follows. Section 2 discusses the concept of taming the complexity of verification/validation solutions. We start with the most general case, static methods, and work toward the most specific, the dynamic solution. Section 3 introduces a method, which makes possible to dynamically validate rule-based systems. Section 4 compares our solution with the related V&V approaches and further highlights the relevance of the suggested approach. Finally, concluding remarks are elaborated.

2 Taming Verification/Validation Complexity

Several static approaches provide formalism and verify that the semantics are preserved or guaranteed during the transformation of a model, e.g. approaches provided by Asztalos et al. [4], Biermann et al. [5], Bisztray and Heckel [6], Cabot et al. [7], or Schatz [8].

The approach of Asztalos et al. focuses on the static analysis of special model processing programs. This approach provides the theoretical basis for a possible verification framework. It applies a final formula that describes the properties that remain true at the end of the transformation. It is possible to derive either proof or refutation of a verifiable property from this final formula. The approach provides predefined components to deduct the desired properties.

In the approach, presented by Bisztray and Heckel, to understand and control the semantic consequences, Communicating Sequential Processes (CSP) are applied to capture the behavior of processes both before and after the transformation. The approach verifies semantic properties of the transformations at the level of rules, such that every application of a rule has a known semantic effect.

In the approach of Bierman et al., model transformations are defined as a special kind of typed graph transformations. The solution implements a formal approach to validate various functional behaviors and consistencies of model transformations.

There exist noticeable differences between the complexity of static and dynamic V&V approaches. The static technique is more general, because its responsibility is to determine if the rule-based system itself meets certain requirements. Contrarily, in the case of the dynamic approach, the transformation is analyzed based on a single specific input.

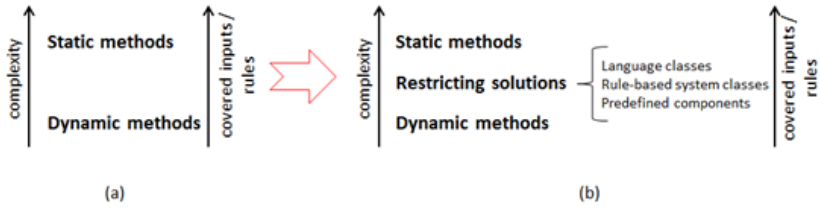

Figure 1
Taming the complexity of verification/validation

It is common knowledge that the algorithmic complexity of V&V can be very challenging, if not altogether hopeless in a general sense. However, practical cases do not require generality. The complexity-related questions are as follows: Does the problem contain specific subclasses that are solvable, yet practically relevant? Is it necessary to analyze the algorithm of the rule-based system? Or will it suffice to verify the system for a certain class of possible input models?

Our classification defines that the static approach is the most general and the dynamic is the most specific of V&V methods (Figure 1a). In order to reduce V&V complexity, we classify the V&V approaches to address complexity. In order to accomplish this, we begin with the general case (static verification) and create more specific cases (dynamic verification). Between these two extreme approaches, we identify several complexity-related restricting solutions (Fig. 1b). These methods do not attempt to prove the semantic correctness for one or all possible inputs (prove the properties of the rule-based system), but instead take a class of input types into consideration. We have identified the following complexity categories:

A. Static methods

B. Restricting solutions:

    1. Language classes

    2. Rule-based system classes

    3. Predefined components

C. Dynamic methods

A. Static methods. Model checker tools (e.g. Augur [9], CheckVML [10], or GROOVE [11]) apply static methods during the verification.

B1. Restricting solutions / Language classes. This approach defines a class of input models. Based on a metamodel, a language class is defined by additional metamodel constraints or the simplification of the metamodel, i.e. through the elimination of some domain concepts. As a result, the modified language contains only a restricted class of original models, therefore, the complexity of the processing transformation decreases along with the complexity of the transformation verification. Examples of language classes are provided in OMG Query/View/Transformation Specification [12]. The Annex A of the QVT specification introduces two language classes: S*imple UML Metamodel* and *Simple RDBMS Metamodel*. These domains provide limited language elements and attributes that are suitable to define the required models, but do not provide additional, unnecessary language constructions. For instance, a *Table* containing *Keys* can be modeled, but the *Key* type does not provide attributes to specify further details. Another example regarding language classes is the limitation of the multiplicity to *1* or *0..1*. A third example presents itself when only finite input models are permitted. A sample language class also introduced in the next section: *DomainServers* (Figure 2).

B2. Restricting solutions / Rule-based system classes. This approach restricts the rule specification language itself. We modify the metamodel of the rule specification language in order to allow for rule-based system definitions with

specific properties. An example of a rule-based system class is one in which rule chains are allowed (successively applying several rules in a predefined order) but loops are forbidden. Proving the termination of such rule-based system requires reasonably less complexity than in the general case, when loops are permitted. For example in the field of layered grammars [13], Bottoni et al. [14] developed a termination criterion which ensures that the creation of all objects of a certain type should precede the deletion of an object of the same type. Therefore, the layer deleting an object of a given type should not create such an object, nor should the subsequent rules. This means the productions in a deletion layer will terminate. Therefore, the termination analysis of transformations satisfying this criterion requires less complexity than the general case.

B3. Restricting solutions / Predefined components. In this case, the verification procedure is constructed from predefined components. We can state facts about the components that treat the verification process as axioms: therefore, the results of other tools or human analysis can be also utilized. Applying these predefined components, we can deduce what output model properties are provided by the given transformation for the provided input domain. For example, the formal language, developed by Asztalos et al. [4], is able to express a set of model transformation properties. The language is appropriate to specify both the properties of the output models and the properties of the relation between the input and output model pairs. In most cases, the proofs within the class of predefined components are conducted by dedicated checker tools (e.g. GROOVE [11] or CheckVML [10]) or through human analysis.

C. Dynamic methods. Examples of dynamic methods are provided by Lengyel [15]. In their approach, the validation of the rule-based system is achieved with constraints assigned to the rules as pre- and postconditions. A similar approach was developed by Narayanan and Karsai [16], in which the semantic equivalence between inputs was guaranteed via bi-simulation checks on the execution log of the transformation.

In applying these restricting solutions, i.e. working with language classes, rule-based system classes, or predefined components, we ensure that (i) the verification of the rule-based system requires less complexity than the classical static verification and (ii) the verification results are valid not only for a specific input, but for a class of input models or rule-based systems. The next section discusses a dynamic validation method.

3 Dynamically Validated Rule-based Systems

There are several model transformation approaches ranging from relational specifications [17] and graph transformation techniques [18], to algorithmic techniques for the implementation of a model transformation. The following provides our categorization of these approaches: [19] traversal-based and direct manipulation approaches [20], template-based approaches (e.g., OCL [21], XPath, or T4 Text Templates), relational approaches (e.g., Query, Views, Transformations (QVT) [12]), graph rewriting-based approaches (e.g., AGG [22], AToM^3 [23], GReAT [24], TGGs [25], VIATRA2 [26], and VMTS [27]), structure-driven approaches (e.g., OptimalJ and QVT), and hybrid approaches that combine two or more of the previous categories (e.g., ATL [28]).

Rule-based systems are often realized based on the graph rewriting-based approach. Therefore, our focus is on the V&V of the graph transformations.

Graph rewriting-based transformations [29] have their roots in classical approaches to rewriting, such as Chomsky grammars and term rewriting [30]. There are many other representations of this, which will be addressed later. In essence, a rewriting rule is composed of a left-hand side (LHS) pattern and a right-hand side (RHS) pattern. Operationally, a graph transformation from a graph G to a graph H is mainly conducted based on the following three steps:

a. Choose a rewriting rule.

b. Find an occurrence of the LHS in host graph G satisfying the application conditions of the rule.

c. Finally, replace the subgraph matched in G by the RHS.

Graph transformations define the transformation of models. The LHS of a rule defines the pattern to be found in the host model; therefore, the LHS is considered the positive application condition (PAC). However, it is often necessary to specify what pattern should not be present. This is referred to as negative application condition (NAC) [31]. Besides NACs, some approaches [22] [26] use other constraint languages, e.g., OCL, Java, C# or Python to define the execution conditions.

The ordering of rules can be achieved by explicit control structures or can be implicit due to the nature of their rule specifications. Moreover, several rules may be applicable simultaneously. Blostein et al. [32] have classified graph transformation organization into four categories. (i) An unordered graph-rewriting system simply consists of a set of graph-rewriting rules. Applicable rules are selected non-deterministically until none are any longer applicable. (ii) A graph grammar consists of rules, a start graph and terminal states. Graph grammars are used for generating language elements and language recognition. (iii) In ordered graph-rewriting systems, a control mechanism explicitly orders the rule

application of a set of rewriting rules (e.g. priority-based, layered/phased, or those containing an explicit control flow structure). (iv) In event-driven graph-rewriting systems, rule execution is triggered by external events. This approach has recently seen a rise in popularity [33].

Controlled (or programmed) graph transformations impose a control structure over the transformation rules to maintain a more strict order of execution in a sequence of rules. The control structure primitives of graph transformation may provide the following properties: atomicity, sequencing, branching, looping, non-determinism, recursion, parallelism, back-tracking and/or hierarchy [15] [30].

Some examples of control structures are as follows: AGG [22] uses layered graph grammars. The layers fix the order in which rules are applied. The control mechanism of AToM^3 [23] is a priority-based transformation flow. Fujaba [34] uses story diagrams to define model transformations. The control structure language of GReAT [24] uses a data flow diagram notation. GReAT also has a test rule construction; a test rule is a special expression that is used to change the control flow during execution. VIATRA2 [26] applies abstract state machines (ASM). VMTS [27] uses stereotyped UML activity diagrams to further specify control flow structures. In [29], a comparative study is provided that examines the control structure capabilities of the tools AGG, AToM^3, VIATRA2, and VMTS.

In the case of rule-based systems, the application order of the rules is supported by a conflict resolution strategy. The strategy may be determined by the actual area or may simply be a matter of preference. In any case, it is vital as it controls which of the applicable rules are fired and thus the behavior of the entire system. The most common strategies are as follows:

a. *First applicable:* If the rules are in a specified order, firing the first applicable rule allows for control over the order in which rules are fired.

b. *Random:* Though it does not provide the predictability or control of the first-applicable strategy, it does have certain advantages. For one, its unpredictability is an advantage in some circumstances (e.g., in games). A random strategy simply chooses a single random rule to fire from the conflict set. Another possibility for a random strategy is a fuzzy rule-based system in which each rule has a factored probability, i.e., some rules are more likely to fire than others.

c. *Least recently used:* Each of the rules is accompanied by a time or step stamp, which marks the time of its last usage. This maximizes the number of individual rules that are fired at least once. This strategy is perfect when all rules are needed for the solution of a given problem.

d. *Best rule:* Each rule is given a weight, which specifies its comparative consideration to the alternatives. The rule with the most preferable outcomes is chosen based on this weight.

3.1 An Example

Rules can be made more relevant to software engineering models if the transformation specifications allow the assigning of validation constraints to the transformation rules.

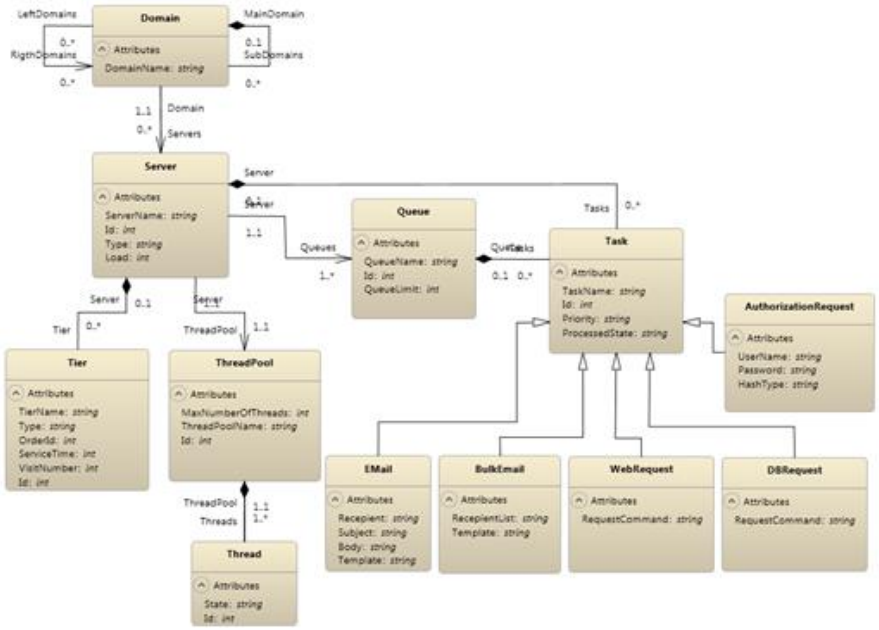

Figure 2
The *DomainServers* metamodel

Figure 2 depicts the metamodel of a domain-specific language. This language defines that an instance model contains *Domain* objects. A domain can contain sub-domains and domains can also be linked to each other. A domain has *Server* objects. A server must belong to a domain. A server contains a *ServerName*, *Id*, *Type* (enum attribute with values *Web*, *Database*, *Mail*, and *Gateway*), and *Load* attributes. *Servers* contain sequentially ordered *Tiers*. A tier has the following attributes: *TierName*, *Id*, *Type* (enum attribute with values *CPU* and *I/O*), *OrderId*, *ServiceTime*, and *VisitNumber*. Each server has exactly one *ThreadPool* element. A *ThreadPool* is comprised of *ThreadPoolName*, *Id*, and *MaxNumberOfThreads* attributes. The *ThreadPool* contains *Threads*. Each thread has *Id* and *State* (enum attribute with values *Ready* and *Occupied*) attributes. *Servers* have one or more *Queues*. A queue has *QueueName*, *Id*, and *QueueLimit* attributes. A queue must belong to a server. Furthermore, servers and queues can contain *Tasks*. *Tasks* assigned to servers are under processing, while tasks in a queue are in waiting state. A task has the following attributes: *TaskName*, *Id*, *Priority* (enum attribute

with values *Normal*, *High*, and *Urgent*), and *ProcessedState* (enum attribute with values *Waiting*, *Processing*, and *Complete*). There are also more specific task types inherited from *Task*: *Email*, *BulkEmail*, *WebRequest*, *DBRequest*, and *AuthorizationRequest*. Each of these metamodel elements also includes further attributes.

Figure 3 introduces a control flow model of a rule-based system. The processing has three transformation rules. The rule *CheckServerLoad* selects a *Server*, which *Load* is over 80%. If there is no such server, then the transformation terminates. Otherwise, a new server node, with a *ThreadPool* and a *Queue* node, is inserted into the domain. Next, the transformation rule, *RearrangeTasks*, rearranges tasks from the queue of the overloaded server to the queue of the new server. The rule *RearrangeTasks* is executed in *Exhaustive* mode: the rule is continuously applied while the *Load* of the overloaded servers is over 70%, and the *Load* of the new server remains under 70%. The transformation is executed in a loop. This means, after easing the load of one server, the process continues and therefore, the transformation can insert additional, new servers.

Figure 3
Example model transformation: *LoadBalancing*

Figure 4 depicts two example rules: *AddNewServer* and *RearrangeTasks*. The figure follows a compact notation, containing no separated LHS and RHS pattern. The colors code the following: black nodes and edges denote unmodified elements, blue ones indicate newly created elements, and red ones mark the elements deleted by the rule. The transformation rule *AddNewServer* gets the *Domain* type node as a parameter and creates the new *Server* with a *ThreadPool*, two *Threads*, a *Tier*, and a *Queue*. The transformation rule, *RearrangeTasks*, receives the two servers with their queues as parameters and performs the rearrangement as a single task. The rule is executed in *Exhaustive* mode, which enables several tasks to be moved between the queues.

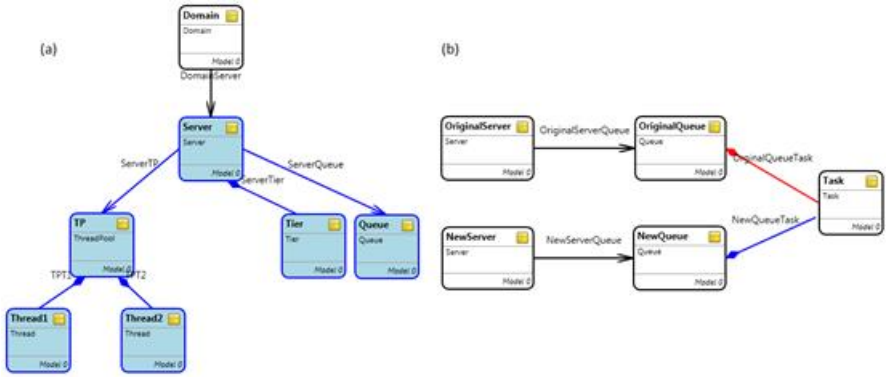

Figure 4
Example model transformation rules: (a) *AddNewServer* and (b) *RearrangeTasks*

Some example constraints assigned to the rules are as follows:

```
context Server inv serverCardinality:
Server.allInstances()->count() < 40

context Queue inv queueCardinality:
Queue.allInstances()->count() >= Server.allInstances()->count()
```

The constraints *serverCardinality* and *queueCardinality* define the number of specific type elements in the model. These are cardinality issues related to the whole model.

```
context Queue inv queueLimit:
QueueLimit < 1500
```

The constraint *queueLimit* is an attribute value constraint that maximizes *QueueLimit* attribute of *Queue* type nodes.

```
context Server inv largeThreadPools:
Server.allInstances->forall(s | s.ThreadPool.Threads->count()<= 50 OR
     (s.Tiers->exists(t | t.Type = Type::CPU) AND
      s.Tiers->exists(t | t.Type = Type::I/O)))
```

The constraint *largeThreadPools* defines that for each server, if the number of threads in the *ThreadPool* exceeds 50, then separated CPU and I/O tiers are employed.

The presented constraints are assigned to the rules and guarantee our requirements. After a successful rule execution, the conditions hold and the output is valid. The fact that the successful execution of the rule guarantees the valid output cannot be achieved without these validation constraints.

3.2 Validating Rule-based Systems

The objective of our research activities is to support the V&V of algorithms performed by rule-based systems. The requirements, assigned to the rules are both input and output related requirements, i.e. we define certain pre- and postconditions that should hold before and after the execution of the rule. In several cases rules do not contain certain node or edge types that are about to be included into our V&V requirements. These requirements may relate to a temporary (during the processing) or a final (following the processing) state of the input or generated models. Moreover, several different directions can be followed; e.g. we can assert additional requirements to the input and output models (metamodel constraints), or the rule-based system can be extended with the use of appropriate testing and validating rules.

Dynamic validation covers both the attribute value and the structure validation, which can be expressed in first-order logic extended with traversing capabilities. Example languages that currently applied for defining attribute value and interval conditions are Object Constraint Language (OCL), C, Java, and Python. These conditions and requirements are pre- and postconditions of a transformation rule.

Definition (Precondition). A precondition assigned to a rule is a Boolean expression that must be true at the moment of rule firing.

Definition (Postcondition). A postcondition assigned to a rule is a Boolean expression that must be true after the completion of a rule.

If a precondition of a rule is not true, then the rule fails without being fired. If a postcondition of a rule is not true after the execution of the rule, the rule fails.

Regarding pre- and postconditions the execution of a rule is as follows (Figure 5):

a. Finding the match according to the LHS structure.

b. Validating the constraints defined in LHS on the matched parts of the input model.

c. If a match satisfies all constraints (preconditions), then executing the rule, otherwise the rule fails.

d. Validating the constraints defined in RHS on the modified/generated model. If the result of the rule satisfies the postconditions, then the rule was successful, otherwise the rule fails.

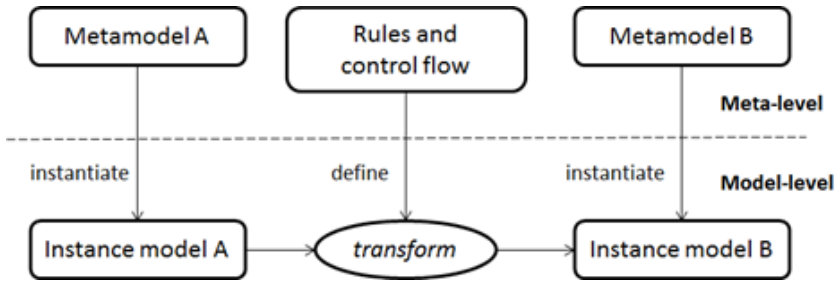

Figure 5
The transformation process

A direct corollary is that an expression in LHS is a precondition to the rule, and an expression in RHS is a postcondition to the rule. A rule can be executed if and only if all conditions enlisted in LHS are true. Also, if a rule finished successfully, then all conditions enlisted in RHS must be true.

Statement 1. If a finite sequence of rules is specified properly with the help of validation constraints, and the sequence of rules has been executed successfully for the input model, then the modified/generated output model is in accordance with the expected result that is described by the finite sequence of transformation rules refined with the constraints [7].

Definition (Low-level construct). Pre- and postconditions defined as constraints and propagated to the rules are low-level constructs.

Definition (High-level construct). Validation, preservation and guarantee properties are high-level constructs.

Definition (Validated rule execution). A rule execution is validated if it satisfies a set of high-level constructs.

To summarize, high-level constructs define the requirements on a higher abstraction level, e.g. servers should not be overloaded. Low-level constructs are the appropriate constraints assigned to the appropriate rules. These constraints assist in achieving the required conditions.

This method can be followed in Figure 4. Finding the structural match the preconditions are validated, and after performing the rule execution, postconditions are validated. Both of the validation should be successful in order for the whole rule to be successful.

With this method the required properties can be defined on low-level, i.e. on the level of rules. In summary, we can say that the presented dynamic approach supports that if the execution of a rule finishes successfully, the generated output is valid and fulfills the required conditions. The validation of the rule-based system is achieved with constraints assigned to the rules as pre- and postconditions.

Statement 2. Rule-based systems can be validated with the presented dynamic validation method.

Statement 3. Taming verification complexity can be applied for rule-based systems.

4 Related Work

In order to underpin the relevance of current results we compiled a collection of challenging transformations requiring V&V. The provided methods support different domain-specific languages-based [35] model-driven approaches.

Giese et al. [36] points out the challenge in using model-driven software development (MDD). The problem is the lack of verified transformations, especially in the area of safety-critical systems. The verification of critical safety properties on the model level is useful only if the automatic code generation is guaranteed to be correct, i.e. the verified properties are guaranteed to hold true for the generated code as well. This means it is necessary to pay special attention to checking for semantic equivalence, at least to a moderate level, between the model specification and the generated code.

In the field of developing safety-critical systems, model analysis possesses advantages over pure testing of implemented systems. For example, important required safety properties of a system under development could be verified on the model level rather than trying to systematically test for the absence of failures.

Narayanan and Karsai [16] have summarized that in the development of a model-based software, a complete design and analysis process involves designing the system using the design language, converting it into the analysis language and performing the verification on the analysis model. They established that graph transformations were a powerful and convenient method increasingly being used to automate this conversion. In such a scenario, the transformation must ensure that the analysis model preserves the semantics of the design model. They concluded that methods are required to verify that the semantics used during the analysis are indeed preserved across the transformation.

de Lara and Taentzer [37] discussed the need for verified and validated model processing in the field of Multi-Paradigm Modeling (MPM) [38]. Software systems have components that may require descriptions using different notations, due to different characteristics. For the analysis of certain properties of the system as a whole, or its simulation, we transformed each component into a common single formalism, in which appropriate analysis or simulation techniques are available.

Varró [39] went on to state that due to the increasing complexity of IT systems and modeling languages, conceptual, human design errors will occur in any model on any high level of the formal modeling paradigm. Accordingly, the use of formal specification techniques alone does not guarantee the functional correctness and consistency of the system under design. Therefore, automated formal verification tools are required to verify the requirements fulfilled by the system model. As the input language of model checker tools is too basic for direct use, model transformations are applied to project behavioral models into the input languages of the model-checking tools.

In conclusion, it is important to understand that model transformations and rule-based systems themselves can be erroneous; therefore, uncovering solutions to make model transformations and rule-based systems free of conceptual errors is essential.

Conclusions

Rule-based systems can effectively automate problem-solving standards. Such systems provide a method for capturing and refining human expertise, and affirm their relevance to the industry. Instead of representing knowledge in a relatively declarative way, i.e., numerous things that are known to be true, rule-based systems represent knowledge in terms of a collection of rules that tell what should be done, i.e., what can be concluded from different situations?

The motivation of the current work was to support the verification/validation of rule-based systems. In this paper, we have introduced the concept of taming verification complexity. We have seen that the static validation method is more general and raises challenges that are more complex. We have discussed the possibilities of reducing the complexity of V&V and have introduced different restricting solutions. Finally, we have presented the dynamic approach, in which the rule-based system is validated for a specific input model.

Then, we have introduced a method, which facilitates to apply the graph rewriting-based dynamic (online) validation results in the field of rule-based systems. The solution facilitates to validate single rules, rule chains, and in effect transformations as a whole. The validation is driven by the pre- and postconditions assigned to these rules.

Our current research activities concentrate on trace-based verification/validation approaches. In these cases, constraints are validated based on the trace files, following the execution. This is the difference between the trace-based approach and the currently presented dynamic approach.

Acknowledgement

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

References

[1] Hayes-Roth, F.: Rule-based systems, Communication of ACM 28, 9, 921-932 (1985), DOI=10.1145/4284.4286 http://doi.acm.org/10.1145/4284.4286

[2] Williams, T. and Bainbridge, B.: Rule-based Systems, In Approaches to Knowledge Representation: an Introduction, Research Studies Press Ltd., Taunton, UK, UK 101-115 (1988)

[3] IEEE Standard Glossary of Software Engineering Terminology, 610.12-1990 (1990)

[4] Asztalos, M., Lengyel, L., Levendovszky, T.: A Formal Framework for Automated Verification of Model Transformations, Software Testing, Verification and Reliability, 23:(5), 405-435 (2013)

[5] Biermann, E., Ermel, C., Taentzer, G.: Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation, Software and Systems Modeling (SoSyM), Springer, 1-24 (2011)

[6] Bisztray, D., Heckel, R., Ehrig, H.: Verification of Architectural Refactorings by Rule Extraction, In Fundamental Approaches to Software Engineering, LNCS, Vol. 4961, Springer, 347-361 (2008)

[7] Cabot, J., Clariso, R., Guerra, E., de Lara, J.: V&V of Declarative Model-to-Model Transformations through Invariants, J. Syst. Softw., Vol. 83(2), 283-302 (2010)

[8] Schatz B.: Formalization and Rule-based Transformation of EMF Ecore-based Models, Software Language Engineering: First International Conference, SLE 2008, France, 227-244 (2008)

[9] Augur website, http://www.ti.inf.uni-due.de/research/augur/index.html

[10] Rensink, A., Schmidt, A., Varró, D.: Model Checking Graph Transformations: A Comparison of Two Approaches. Proceedings of the ICGT 2004: Second International Conference on Graph Transformation, LNCS, Vol. 3256, Springer, Rome, Italy, 226-241 (2004)

[11] GROOVE: GRaphs for Object-oriented VErification Website, http://groove.sourceforge.net/groove-index.html

[12] OMG Query/View/Transformation (QVT) Specification, Meta Object Facility 2.0 Query/Views/Transformation Specification, OMG doc. ptc/07-07-07, 2007, http://www.omg.org/

[13] Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, Sz.: Termination Criteria for Model Transformation, FASE 2005, LNCS, 49-63 (2005)

[14] Bottoni, P., Taentzer, G., Schürr, A.: Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph

Transformation, Proceedings of the Visual Languages 2000 IEEE Computer Society, 59-60 (2000)

[15] Lengyel, L.: Online Validation of Visual Model Transformations, PhD thesis, Budapest University of Technology and Economics, Department of Automation and Applied Informatics (2006)

[16] Narayanan, A., Karsai, G.: Towards Verifying Model Transformations, ENTCS, Vol. 211, 191-200 (2008)

[17] Akehurst, D., Kent, S.: A Relational Approach to Defining Transformations in a Metamodel, In UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, LNCS, Vol. 2460, Springer-Verlag, 243-258 (2002)

[18] Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. editors: Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools, Vol. 2, World Scientific, Singapore (1999)

[19] Mens, T., v. Gorp, P.: A Taxonomy of Model Transformation, Electronic Notes in Theoretical Computer Science, Vol. 152, Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), 125-142 (2006)

[20] Vajk, T., Kereskényi, R., Levendovszky, T., Lédeczi, Á.: Raising the Abstraction of Domain-Specific Model Translator Development, 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, USA, 31-37 (2009)

[21] OMG Object Constraint Language (OCL) Specification, Version 2.2, OMG document formal/2010-02-01, 2010, http://www.omg.org/

[22] AGG: The Attributed Graph Grammar System website, http://tfs.cs.tu-berlin.de/agg

[23] AToM^3: A Tool for Multi-paradigm, Multi-formalism and Meta-modeling website, http://atom3.cs.mcgill.ca

[24] GReAT: Graph Rewriting and Transformation website, http://www.isis.vanderbilt.edu/tools/GReAT

[25] Schürr, A.: Specification of Graph Translators with Triple Graph Grammars, Proceedings of the WG94 international workshop on graph-theoretic concepts in computer science, LNCS, Vol. 903, Springer, Berlin Heidelberg New York, 151-163 (1994)

[26] VIATRA2 (VIsual Automated model TRAnsformations) framework website, http://eclipse.org/gmt/VIATRA2

[27] VMTS: Visual Modeling and Transformation System website, http://www.aut.bme.hu/vmts

[28] ATL: ATLAS Transformation Language website, http://eclipse.org/atl/

[29] Taentzer, G., Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Varró D., Varró-Gyapay, Sz.: Model Transformation by Graph Transformation: A Comparative Study, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (2005)

[30] Rozenberg, G. (ed.): Handbook on Graph Grammars and Computing by Graph Transformation: Foundations, Vol. 1, World Scientific, Singapore (1997)

[31] Habel, A., Heckel, R., Taentzer, G.: Graph Grammars with Negative Application Conditions, Fundamenta Informaticae, Vol. 26, 287-313 (1996)

[32] Blostein, D., Fahmy, H., Grbavec, A.: Issues in the Practical Use of Graph Rewriting, In proceedings of the 5th International Workshop on Graph Grammars and Their App to Computer Science, Williamsburg, USA, LNCS, Vol. 1073, Springer-Verlag, 38-55 (1996)

[33] Guerra, E., de Lara, J.: Event-driven Grammars: Relating Abstract and Concrete Levels of Visual Languages, SoSym, Vol. 6, 317-347 (2007)

[34] Fujaba Tool Suite website, http://www.fujaba.de/

[35] Kövesdán, G., Asztalos, M. and Lengyel L.: Architectural Design Patterns for Language Parsers, Acta Polytechnica Hungarica 11:(5), 39-57 (2014)

[36] Giese, H., Glesner, S., Leitner, J., Schafer, W., Wagner, R.: Towards Verified Model Transformations, In ModeVVa06 (2006)

[37] de Lara, J., Taentzer, G.: Automated Model Transformation and its Validation with AToM3 and AGG, in Diagrammatic Representation and Inference, Lecture Notes in Artificial Intelligence, Vol. 2980, Springer, 182-198 (2004)

[38] de Lara, J., Vangheluwe, H., Alfonseca, M.: Metamodelling and Graph Grammars for Multi-Paradigm Modelling in AToM3, Journal of Software and Systems Modeling, Vol. 3(3), 194-209 (2004)

[39] Varró, D.: Automated Formal Verification of Visual Modeling Languages by Model Checking, Journal on Software and System Modeling, Vol. 3(2), 85-113 (2004)