VERIFICATION OF MODEL TRANSFORMATION

Ákos HORVÁTH
Advisor: Dániel VARRÓ

I. Introduction

As model driven software development (MDSD) pervaded the safety critical (SC) and dependable system development processes, higher demands rose on their design, maintenance and certification. Based on high-level modeling standards (e.g. UML), MDSD separates application logic from underlying platform technology by using platform independent models (PIM) to capture the core functionality of the target system, and platform specific models (PSM) to specify the target system on the platforms (SCJava, ADA, C/C++). PSMs and platform-specific source code are automatically generated from PIM and PSMs, respectively, by using model transformation (MT) techniques.

A critical problem is related to the correctness of model transformations is to guarantee certain semantic properties to hold after transformation execution. For instance, when transforming UML models into Petri nets, the results of a formal analysis can be invalidated by erroneous model transformations as the systems engineers cannot distinguish whether an error is in the design or in the transformation. While there are already a large number of model transformation descriptions [1], we focus on graph transformation [2] (GT) as it is (i) a frequently used mean to capture model transformation and (ii) has thoroughly studied mathematically precise syntax and semantics.

In this paper we introduce our vision for verifying property preservation of graph transformation systems. First, we split the verification process into two levels *design* and *implementation* to separate the precise model from its underlying implementation. The idea is to keep *shape analysis* [3] based abstract interpretation in the design level to verify behavioral attributes for the whole transformation, while on the other hand use Hoare [4] styled code analyzers to verify matching properties of each GT rule used in the transformation. This separation allows to reuse the verification results from the design level in the implementation level in case we can assure the correctness of each used GT rule in the transformation.

The rest of the paper is structured as follows, in Sec. II. we briefly introduce the concept of graph transformation and metamodeling, while Sec. III. proposes our verification approach and finally, Sec. IV. concludes the paper followed by future work.

II. Background

In order to introduce our vision this section briefly introduces the basics of metamodeling and graph transformation.

A. Models and Metamodels

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e., abstract syntax) of modeling languages. More precisely, for the definition of a modelling language the followings shall be given

- the abstract syntax defining the concepts of the given domain and their relations,
- the concrete syntax defining the textual or graphical notations of the concepts,
- well-formness rules defining further constraints for the concepts,
- the formal semantics defining the dynamic behaviour of the models.

In our approach, we use a unified directed graph representation serves as the underlying model of the VIATRA2 [5] framework. This way, graph nodes called entities in VIATRA2 uniformly represent

MOF packages, classes, or objects on different metalevels, while graph edges with identities called relations in VIATRA2 denote MOF association ends, attributes, link ends, and slots in a uniform way. As a summary, nodes represent basic concepts of a (modeling) domain, while edges represent the relationships between model elements.

B. Graph Transformation

Graph transformation (GT) is a rule and pattern-based paradigm frequently used for describing model transformation. A graph transformation rule contains a left-hand side graph LHS, a righthand side graph RHS, and (one or more) negative application condition graphs NAC connected to LHS. A negative application condition is a graph morphism, which maps the LHS pattern to a NAC pattern. In other terms, the LHS and NAC graphs together denote the precondition while the RHS denotes the postcondition of a rule.

The application of a rule to a host (instance) model M replaces a matching of the LHS – which is not invalidated by a matching of the NAC, which prohibits the presence of certain nodes and edges – in M by an image of the RHS.

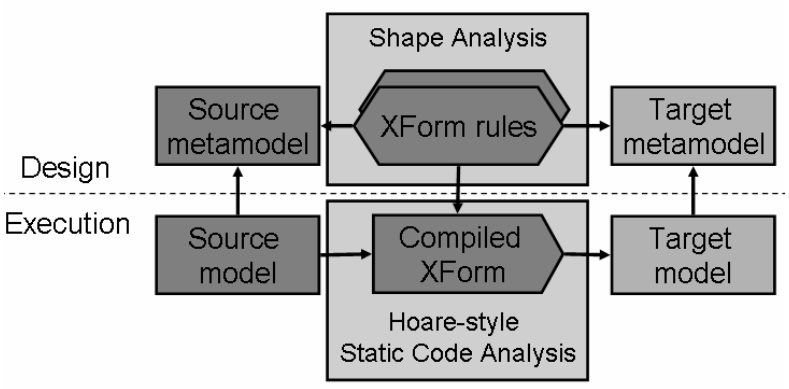

Figure 1: Overview of the approach

Fig 1 gives an overview of our common model transformation process. The model transformation (*XForm rules*) is specified by a number of graph transformation rules. The GT rules are specified with respect to the metamodels of the source and the target metamodel. From these rule specifications, a compiled transformation (*compiled XForm*) is generated. The automatically derived compiled XForm transforms a source model into a target model.

III. Overview of the Approach

Our goal is to verify property preservation for the compiled transformation, meaning that, if a certain property holds in the source model, then after executing the transformation it will also hold in the target model. To do so we separate the verification process into two steps.

- First, we apply shape analysis on the XForm rules to summarize the behavior of a statement on an infinite set of possible rundown states of the GT rules. Shape analysis concerns the problem of determining *shape invariants* for programs that perform destructive updating on dynamically allocated storage. This way correctness of transformation rules applied to *any* model of the specified type can be verified (the concrete instances of the metamodels are irrelevant for the proof).
- Then, as the result of the shape analysis is based on the assumption that the GT rule specification are ”executed” semantically correct, in the second step we focus on the correctness check of the compiled GT rules. As the correctness of the generated compiled code depends on the correctness of the generator itself, which is usually a complex software components that can not

be verified. We use an alternative assurance approach, in which the generator is extended with formal program specification to enable Hoare-style safety analysis for each individually generated GT rule. The crucial step in this approach is to extend the generator to produce all required annotations without compromising the assurance provided by the subsequent verification phase.

A. Design analysis

Shape analysis: In our approach we plan to use the TVLA [6] (Three-Valued-Logic Analyzer), a system for automatically generating a static (shape) analysis implementation from the operational semantics of XForm rules. The small-step structural operational semantics is written in a meta-language based on first-order predicate logic with transitive closure. The main idea is that program states are represented as logical structures and the program transition system is defined using first order logical formulas. TVLA automatically generates the abstract semantics, and, for each program point, produces an abstract representation of the program states at that point. TVLA relies on a fundamental abstraction operation for converting a potentially unbounded structure into a bounded 3-valued structure (logic). 3-valued logic extends boolean logic by introducing a third value 1/2 denoting values that may be 0 or 1. A 3-valued logical structure can be used as an abstraction of a larger 2-valued logical structure. This is achieved by allowing an abstract state (i.e., a 3-valued logical structure) to include summary nodes, i.e., individuals that correspond to one or more individuals in a concrete state represented by that abstract state.

Our initial examples with the TVLA system shows that the mapping of the metamodel to the TVLA is the key for efficient shape analysis generation. On the other hand a surprising experience was that in some case a more precise analysis created smaller shape invariant and thus run faster. Based on [7] this is quite regular in case of 3-valued logic and will have to be extensively studied to create effective mapping of metamodels and GT rules to the analyzing domain.

B. Implementation analyze

Hoare-style platform specific code analyzers: Hoare logic is a formal system to provide a set of logical rules in order to reason about the correctness of computer programs with the rigour of mathematical logic. The central feature of Hoare logic is the *Hoare triple*. A triple describes how the execution of a piece of code changes the state of the computation. A Hoare triple is of the form $\{P\}\ C\ \{Q\}$ where P, Q and C are *precondition*, *postcondition* and *command*, respectively. Based on the concept of pre-/postcondition introduced in the Hoare triple, *design by contract* [8] (DBC or programming by contract) prescribes that software designers should define precise verifiable interface specifications (pre/postconditions) for software components based upon the theory of abstract data types and the concept of a business contract. This means that *contracts* provides semantics to formally describe the behavior of a program module, removing potential ambiguity with regard to the module implementation.

Tools built upon the DBC methodology include the logic of predicate calculus and Dijkstra's weakest precondition calculations. We focused our studies on two of the most widely used frameworks:

- The Spec# [9] programming system is a new attempt having developed at Microsoft Research to extend C# with formally verifiably method contracts in the form of pre-/postconditions as well as object invariants. It consist of (i) the Spec# compiler, which statically enforces non-null types, emits run-time checks for method contracts and invariants, and (ii) the Boogie static program verifier, which generates logical verification conditions from a Spec# program and then uses a built in automatic theorem prover that analyzes the verification conditions to prove the correctness of the program.
- The *KeY* [10] system is a formal software development tool that aims to integrate formal specification, and formal verification of Java programs. The main component of the KeY system is the KeY prover, a semi-automated prover over the Java Dynamic Logic (JavaDL) calculus (with support to Java Modeling Language (JML)). The JavaDL calculus covers the complete

Java Card language, and additionally supports some Java SE features such as multi-dimensional arrays and dynamic object creation. Verification with KeY proceeds by symbolic execution of the Java program being analyzed, where the proof corresponds to a stage during the execution of the program.

Both approaches look promising but does not provide support for: (i) dynamic casting of complex data structures (e.g., arrays), (ii) effective handling of nested loop invariants, (iii) contracts for library functions and finally (iv) user-friendly feedback from proof obligations.

Note that, it is also important to extend the compiled XForm generator in such way that it produces all required annotations (i.e., pre-/postconditions and loop invariants) without compromising the assurance provided by the subsequent verification phase. This is achieved by embedding annotation templates into the code templates, which are then instantiated in parallel by the generator. This is feasible because the structure of the generated code and the possible safety properties are known (from the XForm rules) when the generation is applied. It does not compromise the provided assurance because the annotations only serve as auxiliary lemmas and errors in the annotation templates ultimately lead to unprovable safety obligations.

IV. Conclusion and Future Work

We have presented an ongoing work how graph transformations can be verified with a combination of shape analysis (with TVLA) and static code analyzer (e.g., Spec#, KeY). In the current state of our research, we have studied the boundaries of Hoare-style static code analyzers with respect to complex object navigation (as being the core of transformation implementation). It resulted in state space explosion in case of common implementations of GT rules and have to be further studied to achieve analyzable implementation.

As for the future, we plan to finish formalizing GT rules in 3-valued logic to achieve feasible shape analyze results. Current research in this direction shows that in case of strict metamodels the shape analysis resulted suitable shape invariants for the automatic property checks.

References

[1] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003.

[2] G. Rozenberg, Ed., *Handbook of Graph Grammars and Computing by Graph Transformation, volume 1: Foundations*, World Scientific, 1997.

[3] M. Sagiv, T. Reps, and R. Wilhelm, "Parametric shape analysis via 3–valued logic," in *Symposium on Principles of Programming Languages*, pp. 105–118. ACM Press, 1999.

[4] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, 12(10):576–580, 1969.

[5] A. Balogh and D. Varró, "Advanced model transformation language constructs in the VIATRA2 framework," in *Proc. of the 21st ACM Symposium on Applied Computing*, pp. 1280–1287, Dijon, France, April 2006. ACM Press.

[6] T. Lev-Ami, R. Manevich, and S. Sagiv, "Tvla: A system for generating abstract interpreters," in *IFIP Congress Topical Sessions*, R. Jacquart, Ed., pp. 367–376. Kluwer, Aug. 2004.

[7] R. Wilhelm, S. Sagiv, and T. W. Reps, "Shape analysis," in *Computational Complexity*, pp. 1–17, 2000.

[8] B. Meyer, "Applying "design by contract"," 25(10):40–51, Oct. 1992.

[9] "Spec#, The Spec# programming system," http://research.microsoft.com/specsharp/.

[10] "The KeY Project, Integrated Deductive Software Design," http://www.key-project.org/.