# SyVOLT: Full Model Transformation Verification Using Contracts

Levi Lúcio*, Bentley James Oakes*, Cláudio Gomes†,
Gehan M. K. Selim‡, Juergen Dingel‡, James R. Cordy‡ and Hans Vangheluwe†,*
*School of Computer Science, McGill University, Canada
levi@cs.mcgill.ca, bentley.oakes@mail.mcgill.ca
†MSDL Lab, University of Antwerp, Belgium
claudio.goncalvesgomes@ua.ac.be, hans.vangheluwe@ua.ac.be
‡School of Computing, Queen's University, Canada
{gehan, cordy, dingel}@cs.queensu.ca

***Abstract*—We introduce SyVOLT, a plugin for the Eclipse development environment for the verification of structural pre-/post-condition contracts on model transformations. The plugin allows the user to build transformations in our transformation language DSLTrans using a visual editor. The pre-/post-condition contracts to be proved on the transformation can also be built in a similar interface. Our contract proving process is exhaustive, meaning that if a contract is said to hold, then the contract will hold for all input models of a transformation. If the contract does not hold, then the counter-examples (i.e., input models) where the contract fails will be presented.**

**Demo: https://www.youtube.com/watch?v=8PrR5RhPptY**

## I. Introduction

Model transformations are at the center of model-driven development, making pragmatic and usable tools for their verification indispensable. In this paper we introduce SyVOLT (Symbolic Verifier of mOdeL Transformations) [5], an Eclipse plugin that allows verifying pre-/post-condition structural contracts on model transformations. SyVOLT's operation relies on a theoretical framework that has been developed for the DSLTrans model transformation language. In this framework, pre-/post-condition contracts can be shown to either hold for all input/output pairs resulting from executing a given DSLTrans model transformation, or not to hold for at least one of those input/output pairs [16].

Extensive work exists on the verification of different aspects of model transformations [7]. In [26] the authors describe a method where 'Tracts' can be specified for model transformations. Tracts resemble SyVOLT's contracts and define a set of constraints on the source and target metamodels, a set of source-target constraints, and a tract test suite, i.e., a collection of source models satisfying the source constraints. The accompanying TractsTool can then automatically test the transformation by verifying if all source/target model pairs satisfy the constraints in the tract test suite. Several other approaches support the testing of model transformations based on different kinds of contracts such as model fragments [18], graph patterns [13], [9], Triple Graph Grammars (TGGs) [27] or dedicated testing languages [12].

The tool described in [11] automatically translates transformations in a number of transformation languages (such as ATL) into OCL. Verification is then done using a model finder which will attempt to find a counter-example to the property being proved. Anastasakis et al. [8] translate model transformations to Alloy in order to verify if given assertions hold for the given transformations. Both these approaches attempt to exhaustively verify the model transformation, as SyVOLT does. However, while SyVOLT guarantees that all counter-examples to a contract are always found, this is not granted by the two approaches introduced above.

The distinguishing feature of SyVOLT is that, unlike the methods described above, our tool allows for contracts to be proved for all possible transformation executions, i.e., for all possible input models. However, we share with most tools described above the same implication idea: the pre-condition of a contract sets constraints on the input models to the transformation, and the post-condition defines constraints on the output model.

## II. Highlights

SyVOLT has a number of unique features, outlined below.

### A. Input Independence and Exhaustiveness

SyVOLT proves that pre-/post-condition contracts hold for a model transformation. Such contracts establish relations between patterns occurring in input and output models of a model transformation. If a contract holds, a formal guarantee exists that whenever a transformation's input model contains the pattern specified in the pre-condition of the contract, the transformation's output model will contain the pattern specified in the contract's post-condition. Contracts can optionally include traceability relations between input and output patterns. Our technique is exhaustive and input-independent, in the sense that whenever a contract holds, it will hold for all possible input models for that transformation. This is possible because SyVOLT operates on specifications of out-place model transformations, where unbounded loops and model element deletions are not allowed. A discussion on the soundness and completeness of our approach is provided in [16].

```
> Proving contracts:
> Contract ``daughterMother'' holds for all input models!
> Contract ``motherFather'' does NOT hold for all input
models! The contract fails on the following Path Conditions:
['EmptyPathCondition_RootRule_FatherRule_MotherRule', ...]
> The smallest Path Conditions where the contract fails are:
['EmptyPathCondition_FatherRule_MotherRule']
> Time to verify 2 contracts: 11.6834638966 seconds.
```

Fig. 1. Sample output of the contract prover

### B. Integration with Eclipse / Graphical Modelling

Eclipse is a popular development environment, as many model transformation tools such as ATL, DSLTrans [10] and EGL [3] are integrated with the Eclipse Modeling Framework (EMF) [2]. To take advantage of this ecosystem, SyVOLT integrates with EMF to represent models in a multitude of syntaxes, from graphical to textual. Modellers may then operate in their preferred syntax, although the authors suggest the visual representation of a contract in the SyVOLT editor allows for intuitive understanding of the contract's meaning.

### C. Push-Button Proofs

The proving process for a SyVOLT contract is fully automatic and all of the approach's formal details are completely hidden from the user. Once the transformation and the contracts of interest are created, one command will start the property proving process. This process will automatically create all required artifacts (as detailed in Section III), run the process, and provide the results to the user within the Eclipse environment. This allows the user to continually stay within the Eclipse environment, where he or she develops the contracts and the model transformations.

### D. Counter-Examples

When a given contract does not hold on a given model transformation, SyVOLT can produce additional information for the user to pinpoint where the contract's violation occurs. This information is in the form of the set of model transformation rules used to build a particular path condition for which the contract fails. A counter-example is any input model where this set of rules would execute. For example, the sample output in Figure 1 alerts the user that the contract *motherFather* will fail when only the *mother* and *father* rules execute in the transformation.

### E. Based on Symbolic Execution

Our technique shares its principles with symbolic execution, a classic method to verify code. The underlying idea entails building a finite representation of the (infinite) set of computations that can be expressed by a model transformation specification. In this context, each symbolic execution – which in the context of our work we call a *path condition* – is an overlapping combination of a subset of the transformation's rules. Because a path condition contains a number of rules, it represents the execution of the model transformation over any input model those rules match on.

Contracts of interest are proved on the set of path conditions built for a model transformation, and are extrapolated to the infinite set of the model transformation's computations through an abstraction relation [16].

### F. Proving Contracts about ATL Model Transformations

The Atlas Transformation Language (ATL) [1] is commonly-used in both industrial and academic applications. In order to enable contract proving on ATL transformations, we have developed a higher-order transformation that is able to automatically transform declarative ATL transformations into DSLTrans transformations [19]. In the future we will integrate this higher-order transformation into SyVOLT's user interface.

### G. Scalability and Speed

We have some evidence that SyVOLT scales to transformations of practical interest. In particular we have verified contracts on DSLTrans transformations with up to over 60 rules, and ATL transformations with up to 13 rules [19]. From our own experience, the size of a DSLTrans transformation ranges from 10 up to 50 rules, while the average size of an ATL transformation is around 20 rules [15]. Even though our technique is exhaustive, our experiments show that verification can be performed within seconds. Gehan Selim's PhD thesis [22] provides further evidence of SyVOLT's speed, by verifying a relatively large model transformation for giving semantics to the UML-RT language in terms of the Kiltera process language [20]. SyVOLT's symbolic execution engine is fully homegrown [17] and does not depend on third-party solvers. Although this has implied a large effort to build the codebase, it has allowed us to have the required control over the code to iteratively optimize the engine for space and time economy. [23] demonstrates that our prover is substantially faster than similar approaches based on SAT solvers.

## III. Architecture

The guiding principle of the Modelling, Design, and Simulation Lab is that all tools and processes should be explicitly modelled at an appropriate abstraction level. This practice ensures that software development effort is targeted at the problem's essential complexity, while alleviating complexity induced by the computational platforms used.

SyVOLT's codebase has been developed by applying these principles. We have used available model-driven development technology as much as possible, both to develop and as a part of SyVOLT itself. In particular, we have made it such that the DSLTrans transformations, the SyVOLT contracts, and path conditions are all represented and treated as models by the proof engine. Moreover, we have operationally encoded the operations required by the algorithm for building a contract proof as model transformations [17].

The following model-driven development tools have been used in SyVOLT's development:

- **Himesis** [21]: Himesis is a typed graph representation format, built upon the open-source igraph library [6]. In [25], it is reported empirically that Himesis is a good format to perform the typical graph manipulation

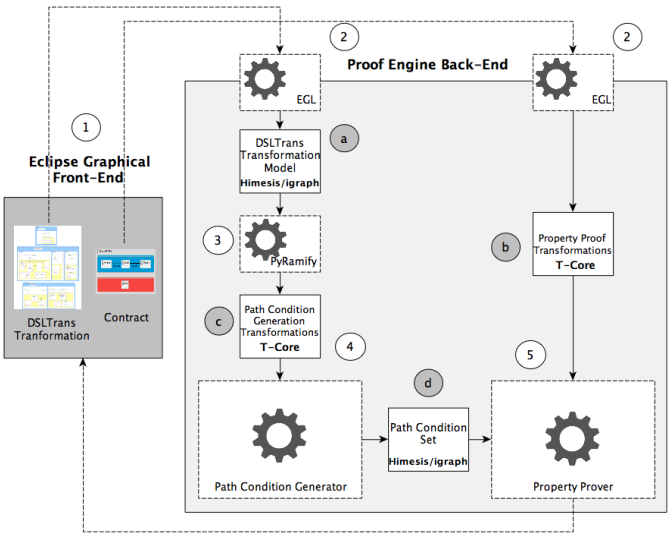


Fig. 2. The architecture of the SyVOLT tool

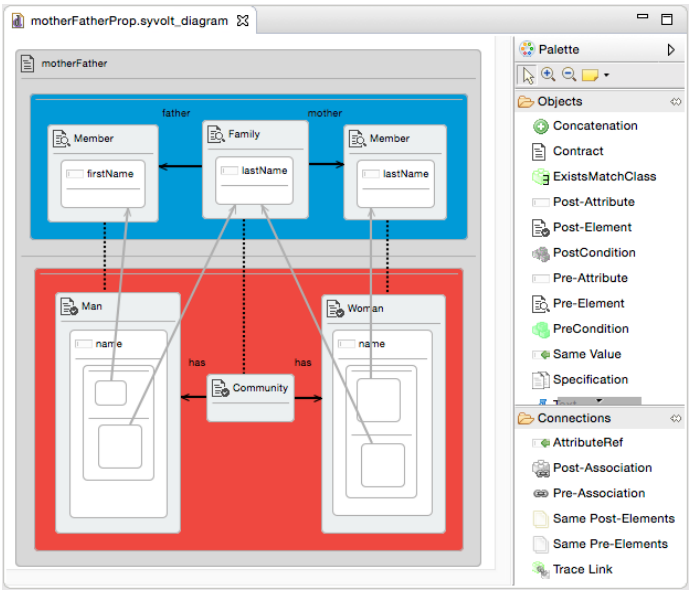


Fig. 3. The SyVOLT editor within Eclipse

operations. Himesis is used pervasively within SyVOLT to represent all models and model transformation rules required by the proof algorithm.

- **T-Core** [24]: T-Core is a collection of model transformation primitives allowing fine-grained manipulation of models represented in the Himesis format. The main operations of T-Core are model *matching*, model *rewriting* and *iterating* through a set of match sites in a model. The level of control in model manipulation, together with T-Core's speed and scalability when treating large models, suited our needs well when implementing the property proof algorithm described in [16]. Note that because T-Core is also explicitly modelled, a T-Core model transformation rule is also a (Himesis) model.
- **Eclipse Modelling Framework** (EMF): SyVOLT makes use of EMF's Ecore format for the XMI representation of DSLTrans transformations and SyVOLT's contracts within the Eclipse editors.
- **Epsilon Generation Language** (EGL): Converting Ecore models into Himesis models is achieved using EGL, a model-to-text transformation language.

Figure 2 shows the architecture of our tool, where squares containing gears (and annotated with number identifiers) represent computations and squares containing no gears (and annotated with letter identifiers) represent produced and consumed data. Two essential blocks are depicted: the graphical Eclipse front-end for user interaction and the proof engine back-end which implements the proving algorithm. The front-end and the back-end communicate in the following way: in the left-to-right direction, a number of artifacts (models and model transformations) are synthesized from the graphical representations of the DSLTrans transformations and of the SyVOLT contracts, and passed to the back-end. In the reverse direction, the proof result and counter-examples, if any, are passed from the proof engine to the front-end.

In what follows, we will briefly visit each computational component of SyVOLT to describe its function and the technologies employed in its development.

SyVOLT's architectural components described in Figure 2 are orchestrated by an ANT script from within Eclipse. This script makes sure all components communicate and execute in the right order, and allows contract proof to run fully automatically at the push of a button.

### *1. SyVOLT Contract Editor*

The SyVolt contract editor, shown in Figure 3, is realized by a set of Eclipse Graphical Modeling Framework (GMF) [4] plugins, generated using EuGENia [14]. The user can prescribe contracts (the gray, red and blue rectangle in the left-hand side of Figure 3) and propositional logic properties involving contracts using the toolbox (right-hand side of Figure 3).

Internally, the graphical editor reads and write two models: the domain model and the graphical model. The graphical model can be seen as the realization of the concrete syntax by storing the coordinates and other visual information about the shapes that are shown in the graphical editor (Figure 3). In contrast, the Ecore domain model contains the abstract syntax: the essence of the elements that form the contracts and the propositional properties. The domain model is the artifact used when generating the models for the construction of the proof.

### *2. Generating Rule and Contract Artifacts*

The translation of DSLTrans transformations and SyVOLT contracts into Himesis models and model transformation rules was achieved using the EGL. Specifically, two EGL model-to-text transformations were used: one to generate the Himesis models representing a DSLTrans transformation (marked (a) in Figure 2); and another one to generate the T-Core model transformation rules necessary to prove SyVOLT contracts (marked (b) in Figure 2).

From the Ecore models of the DSLTrans model transformation and contracts of interest, the EGL transformations produce a number of Python classes that inherit from the Himesis library and that represent models and T-Core transformation rules. These classes will be subsequently loaded in memory and manipulated by the proof engine back-end.

### *3. Generating Artifacts for Path Condition Generation*

Additional artifacts need to be generated for the property proof algorithm to execute. These are the path condition generation T-Core model transformations (marked (c) in Figure 2) that are needed to perform the model manipulations for a set of path conditions. These model transformation blocks allow combining the rules of a DSLTrans model transformation into a set of path conditions, as per the algorithm described in [16].

This additional model transformation generation step is achieved by the PyRamify component. PyRamify is a Python script that takes as input Himesis graphs representing the rules in the transformation. It produces T-Core matchers and rewriters for each rule. While matchers allow the contract prover to determine if and how rules might combine with a path condition under construction, rewriters combine the right-hand side of a DSLTrans rule with that same path condition.

Currently, PyRamify is implemented as a Python script, as we were unable to implement automatic creation of higher-order artifacts (in our case, T-Core transformations) using graph-based model transformations. This indicates a need to further develop model-driven engineering tools that allow for the explicit manipulation of model transformations as models.

### *4. Path Condition Generation*

Once the required artifacts have been produced, the prover moves onto the path condition generation step. The path condition generation algorithm starts from the empty path condition, representing the case where no rules in the transformation have executed. Then, each rule in the DSLTrans model transformation under analysis is combined with the path conditions generated thus far, the using path condition generation T-Core matchers and rewriters (marked (c) in Figure 2). As each rule is considered, in the same order it would occur during the transformation's execution, the set of path conditions will grow to represent all allowed rules combinations.

### *5. Contract Proof*

The *contract prover* component requires two inputs: the path conditions set, built by the path condition generator (marked (d) in Figure 2); the contract proof T-Core transformations (marked (b) in Figure 2). Pre/post condition contracts form an implication, which needs to be checked for each path condition. For a contract to hold on a DSLTrans model transformation, the contract's implication should hold for every path condition generated for that model transformation. Thus, for each submodel of the path condition that is isomorphic to the pre-condition's model, the algorithm will try to find a submodel of the path condition that is isomorphic to the post-condition's model. T-Core matchers from the contract proof T-Core transformations allow checking the existence of these relations between the contract and the post-condition.

## REFERENCES


[1] Atlas Transformation Language. http://eclipse.org/atl.

[2] Eclipse Modelling Framework. https://eclipse.org/modeling/emf/.

[3] Epsilon Generation Language. http://www.eclipse.org/epsilon/.

[4] Graphical Modeling Project. http://www.eclipse.org/modeling/gmp/.

[5] SyVOLT tool. http://msdl.cs.mcgill.ca/people/levi/contractprover.

[6] The igraph Network Analysis Package. http://igraph.org/.

[7] M. Amrani, L. Lúcio, G. M. K. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *ICSTW*, pages 921–928, 2012.

[8] K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVa*, 2007.

[9] A. Balogh et al. Workflow-Driven Tool Integration Using Model Transformations. In *Graph Transformations and Model-Driven Engineering*, pages 224–248. Springer, 2010.

[10] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. Dsltrans: A turing incomplete transformation language. In *Software Language Engineering*, pages 296–305. Springer, 2011.

[11] F. Büttner, M. Egea, E. Guerra, and J. De Lara. Checking model transformation refinement. In *Theory and Practice of Model Transformations*, pages 158–173. Springer, 2013.

[12] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: A Unit Testing Framework for Model Management Tasks. In *MODELS*, pages 395–409. Springer, 2011.

[13] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated verification of model transformations based on visual contracts. *Autom. Softw. Eng.*, 20(1):5–46, 2013.

[14] D. Kolovos, L. Rose, S. Abid, R. Paige, F. Polack, and G. Botterweck. Taming EMF and GMF Using Model Transformation. In D. Petriu, N. Rouquette, and O. y. Haugen, editors, *Model Driven Engineering Languages and Systems SE - 15*, volume 6394 of *Lecture Notes in Computer Science*, pages 211–225. Springer Berlin Heidelberg, 2010.

[15] A. Kusel, J. Schoenboeck, M. Wimmer, W. Retschitzegger, W. Schwinger, and G. Kappel. Reality check for model transformation reuse: The ATL transformation zoo case study. In *Proceedings of the Second Workshop on the Analysis of Model Transformations (AMT 2013), Miami, FL, USA, September 29, 2013*, 2013.

[16] L. Lúcio, B. Oakes, and H. Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Technical report, Technical Report SOCS-TR-2014.1, McGill U, 2014.

[17] L. Lúcio and H. Vangheluwe. Model Transformations to Verify Model Transformations. In *Online Proceedings of Verification of Model Transformations (VOLT)*, 2013.

[18] J.-M. Mottu, B. Baudry, and Y. L. Traon. Model transformation testing: oracle issue. In *ICSTW*, pages 105–112. IEEE, 2008.

[19] B. J. Oakes, J. Troya, L. Lucio, and M. Wimmer. Fully verifying transformation contracts for declarative atl. Accepted for publication at MoDELS 2015.

[20] E. Posse and J. Dingel. An executable formal semantics for UML-RT. *Software and Systems Modeling*, pages 1–39, 2014.

[21] M. Provost. Himesis: A hierarchical subgraph matching kernel for model driven development. In *Masters Abstracts International*, volume 45, 2006.

[22] G. M. Selim. *Formal Verification of Graph-Based Model Transformations*. PhD thesis, Queen's University, 2015.

[23] G. M. Selim, L. Lúcio, J. R. Cordy, J. Dingel, and B. J. Oakes. Specification and verification of graph-based model transformation properties. In *Graph Transformation*, pages 113–129. Springer, 2014.

[24] E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *Electronic Communications of the EASST*, 29, 2010.

[25] E. Syriani and H. Vangheluwe. Performance analysis of Himesis. Technical report, 2010.

[26] A. Vallecillo, M. Gogolla, L. Burgueno, M. Wimmer, and L. Hamann. Formal specification and testing of model transformations. In *Formal Methods for Model-Driven Engineering*, pages 399–437. Springer, 2012.

[27] M. Wieber, A. Anjorin, and A. Schürr. On the Usage of TGGs for Automated Model Transformation Testing. In *Theory and Practice of Model Transformations*, pages 1–16, 2014.