

Model Transformations to Verify Model Transformations

Levi Lúcio^aHans Vangheluwe^ba. McGill University, Montreal, QC, Canada (levi@cs.mcgill.ca)b. University of Antwerp, Antwerp, Belgium and
McGill University, Montreal, QC, Canada (hv@cs.mcgill.ca)

Abstract In this paper we present a novel technique and a prototype implementation for proving properties of model transformations expressed in the DSLTrans language. The approach is based on symbolic execution and the properties we are interested in concern relations between the structure of the input and output models. In particular, the properties are implications of the form ‘if a structural relation between some elements of the source model holds, then another structural relation between some elements of the target model must also hold’. Our technique is transformation dependent but model independent, meaning the proofs we produce hold for all executions of a given DSLTrans transformation specification running on any instance of the transformation’s input metamodel. Our proof technique is based on (i) building the finite symbolic execution for a given DSLTrans transformation and (ii) on checking whether the property holds for all elements of the finite symbolic execution execution set. We explain how to build a symbolic execution and a proof by using model transformations written in our T-Core framework and finish with some experimental performance results.

1 INTRODUCTION

Model transformations are one of the main enablers of Model Driven Development (MDD) [1]. Mens *et al.* have called for the development of verification, validation and testing techniques for model transformations in 2006 [2]. Despite the many publications on this topic since then, the field of analysis of model transformations seems to be still in its (late) infancy, as evidenced by [3].

The research presented here follows from our proposals in [4] and [5]. In [4] we introduced the DSLTrans transformation language. DSLTrans is Turing Incomplete, as it avoids constructs which imply unbounded recursion or non-determinism. Despite this expressiveness reduction, we have shown via several examples [6, 7, 8] that DSLTrans is sufficiently expressive to tackle typical translation problems.

Our verification transformation technique is based on the theory introduced in [5], where we described how to abstractly build a symbolic execution for DSLTrans transformations. Additionally, in [5], we mathematically proved that such a symbolic execution is finite, given an abstraction over the number of times the transformation’s rules match on concrete elements of input models. This finiteness a necessary condition for our technique to be applicable.

The properties we prove are *model syntax relations* [3]. Such properties are expressed as in the literature as precondition-postcondition axioms involving statements

about the syntactic structure of the input and output models of a transformation. Several authors [9, 10, 11] have explored these properties using techniques distinct from ours. According to the classification presented in [3], our technique is *transformation dependent* and *input independent*, meaning we can prove properties that hold for all executions of a given model transformation, irrespective of the input model.

The paper is organised as follows: section 2 introduces the DSLTrans model transformation language; in section 3 we present our property language; sections 4 and 5 describe respectively the algorithms for symbolic execution construction and for property proof; section 6 presents the implementation of our technique and performance results; finally, in section 7 we conclude with our contributions and future work.

2 The DSLTrans Transformation Language

DSLTrans is a graph-based transformation language and as such shares its principles with transformation languages and tools such as AGG [12], AToM³ [13] and VIATRA2 [14]. The language and transformation engine have been implemented as an Eclipse plugin [7]. DSLTrans computes transformations of models defined as instances of EMF metamodels. A transformation is composed of a list of layers that are executed sequentially. A layer is a set of transformation rules, which produces a deterministic result irrespective of rule execution order. Each transformation rule is a pair $(MatchModel, ApplyModel)$, where *MatchModel* is a pattern holding elements and associations from the source metamodel, and *ApplyModel* is a pattern holding elements and associations of the target metamodel. Figure 1 depicts an example DSLTrans transformation rule. Rule execution implies writing in the output model one instance of the *ApplyModel* pattern per instance of the match pattern found.

DSLTrans’ transformations are strictly outplace, meaning no changes are allowed to the input model. The output metamodel for a DSLTrans transformation can however be the same as the input metamodel. Also, elements cannot be removed from the output metamodel as the result of applying a DSLTrans rule. This restriction is consistent with the usage of model transformations as translations, as no deletion of output elements is strictly required. Such restriction is however not compatible with expressing simulations of reactive systems as model transformations. This illustrates the boundaries of the applicability of DSLTrans and that expressiveness reduction entails a compromise with the class of problems that can be tackled.

In addition to the source and target metamodel patterns used respectively on the *MatchModel* and *ApplyModel* of a transformation rule, DSLTrans also allows indirect links on the match part of a rule. Indirect links match transitively over acyclic EMF containment associations. An example of the concrete syntax of indirect links can be seen in figure 1 as the dashed horizontal line connecting the *MatchModel* elements.

A distinctive feature of DSLTrans is the use of backward links. Backward links connect elements of the match and the apply patterns of a DSLTrans’ rule and allow referring to the traces of the execution of rules in a previous layer. They can be seen in

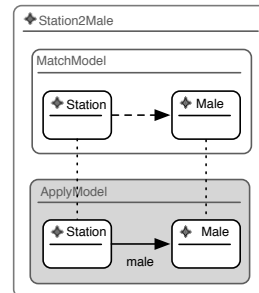


Figure 1: DSLTrans Rule Example

figure 1 as the dashed vertical lines connecting *MatchModel* to *ApplyModel* elements in the rule. As the input model always remains intact, elements created by rules of a previous layer can only be referred to using backward links. Finally, DSLTrans allows negative conditions over match pattern elements, match associations, indirect links and backward patterns. Also, a small language exists for building attributes of apply model elements from references to one or more *MatchModel* element attributes.

Given DSLTrans’ expressiveness, we can mathematically guarantee that all transformations expressible in DSLTrans are both *terminating* and *confluent* [4]. DSLTrans is, to the best of our knowledge, the only graph based transformation language where *termination* and *confluence* are enforced by construction.

3 DSLTrans Model Syntax Relation Properties

As mentioned previously, the properties we are interested in proving about DSLTrans model transformations are in the form of precondition-postcondition axioms. The preconditions and postconditions are syntactic constraints on the input and output models of the DSLTrans transformation being analysed. Preconditions and postcondition constraints are expressed as patterns, primarily as is done respectively in the *MatchModel* and *ApplyModel* patterns of DSLTrans transformation rules. Preconditions use the same pattern language as the *MatchModel* part of DSLTrans rules, involving the possibility of expressing several occurrences of the same metamodel element and indirect links. Indirect links in properties have the same meaning as in the *MatchModel* part of DSLTrans rules – they involve patterns over the transitive closure of containment links in input models. Postconditions also use the same pattern language as the *ApplyModel* patterns of DSLTrans transformation rules, with the additional possibility of also expressing indirect links for patterns including the transitive closure of containment links in output models. Backward links can also be used in properties to impose traceability relations between precondition and postcondition elements. A formal definition of our property language can be found in [5].

4 Symbolic Execution Construction

In order to explain the concept of symbolic execution of a DSLTrans transformation, let us make an analogy with program symbolic execution as introduced by King in his seminal work “Symbolic Execution and Program Testing” [15]. According to King, a symbolic execution of a program is a set of *constraints* on that program’s *input variables* called *path conditions*. Each *path condition* describes a traversal of the conditional branching commands of that program. A *path condition* is symbolic in the sense it *abstracts* as many concrete executions as there are instantiations of the path condition’s variables that render the path condition’s constraints true. The correspondence between the symbolic execution concepts for *programs* and *model transformations* is shown in table 1.

The construction of a symbolic execution for a DSLTrans transformation begins by building the powerset of all the rules in the first layer of a transformation. This means that we are building all the possible combinations of applications of rules in the first layer. Each such rule combination represents a symbolic execution of the first layer. To continue the analogy with program symbolic execution we will henceforth refer to any combination of rules of a DSLTrans transformation as a *path condition*. The fact that in DSLTrans the rules within a layer can be executed in any order with a deterministic result allows us to consider only a fraction of the path conditions that would be necessary than if order would be relevant.

	Program Symbolic Execution	Transformation Symbolic Execution
Abstraction Over	Sets of data values	Sets of models (graphs)
Input Variables	Programming language variables (int, string, float, etc)	Metamodel classes, relations and attributes
Constraints	Predicates on variables imposed by assignment or conditional statements	Metamodel patterns imposed by transformation rules
Path Conditions	Conjunction of predicates on variables	Conjunction of metamodel patterns

Table 1: *Program and Model Transformation Symbolic Execution*

Having produced the symbolic execution for the first layer, or for the layers 1 through l , we can now proceed to layer $l + 1$. As before, we calculate the powerset of the rules in layer $l + 1$. However, we now need to understand how each one of these newly built path conditions affects each partial path condition built for layer l . When we analyse a path condition belonging to the powerset of layer $l + 1$ (noted in what follows PC_{l+1}) against a path condition belonging to the symbolic execution built by the rules of layer 1 through l (noted in what follows $PC_{1..l}$), several cases may occur:

1) if none of the rules in the PC_{l+1} contains backward links, a new path condition is added to the symbolic execution by extending $PC_{1..l}$ with the union of $PC_{1..l}$ and PC_{l+1} . This union is built adding the rules in PC_{l+1} to the rules in $PC_{1..l}$;

2) if the rules in PC_{l+1} include backward links, we need to analyse if those backward links correspond to traces between match and apply elements generated by rules in $PC_{1..l}$. If this is not the case the conditions for at least one of the rules from PC_{l+1} to execute are satisfied and PC_{l+1} cannot be added to the symbolic execution;

3) if the rules in PC_{l+1} include backward links and all those backward links correspond to traces between match and apply elements generated by rules in $PC_{1..l}$ then, as in case 1, a new path condition can be added to the symbolic execution. This new path condition is formed differently than in case 1: all rules from PC_{l+1} containing backward links are merged with the rules from $PC_{1..l}$ where the traces corresponding to the backward links were generated. Additionally, $PC_{1..l}$ is removed from the symbolic execution. As all elements necessary for rules of PC_{l+1} including backward links were generated by the rules of $PC_{1..l}$, the rules from PC_{l+1} with backward links necessarily execute. As such, $PC_{1..l}$ can no longer exist on its own in the symbolic execution;

4) a slight variation of case 3 is where more than one backward link from the same rule in PC_{l+1} is matched over the same trace of a rule from $PC_{1..l}$. In this case, in addition to what happens in case 3, $PC_{1..l}$ also needs to be kept in the symbolic execution. This is because the instantiation relation does not distinguish the number of instances of a metamodel element.

The full description of the symbolic execution construction algorithm is in [16].

5 Property Proof

After the symbolic execution is built, the property proof is performed as follows:

1) A path condition is taken from the symbolic execution. If no more path conditions exist, the property holds. A first check is done to decide if the path condition under consideration has all the metamodel match classes expressed in the property. If so, point 2) is executed, otherwise point 1) is executed with a new path condition;

2) Because we do not know whether two match elements of the same type occurring in two different rules in the same path condition consume the same or different instances in a concrete input model, we need to consider two cases: (i) the case where those two distinct match elements from different rules consume two different instances of that type in the input model; and (ii) the case where those two match elements consume the same instance in the input model. This is achieved by building for a given path condition all the possibilities of collapsed elements of the same type belonging to different rules. A thorough description of the collapsing algorithm can be found in [16]. We then iterate over the path conditions resulting from the collapse operation. For each of those path conditions we check whether the match part of the property is a subgraph of the path condition, in which case point 3) is executed for the collapsed path condition under analysis. Finally, we go back to point 1);

3) We check if the whole property is a subgraph of the collapsed path condition. If this is not the case then the property does not hold and the path condition itself serves as a counterexample for the property. Otherwise, we go back to point 2).

6 Implementation and Results

We have built a prototype implementation of the algorithms presented in sections 4 and 5 using our T-Core model transformation framework [17]. The implementation can be found under [18] as a Python package. All the operations in our algorithms involving DSLTrans’ rule and property manipulations were built as T-Core model transformations. Rules and properties were metamodeled and modeled using AToM³ [13]. Python was used as a scheduling language such that the control flow in the symbolic execution construction and property proof algorithms presented in sections 4 and 5 can be achieved. Given that many similar situations have to be investigated during symbolic execution construction and property proof, memoisation was used whenever possible to avoid isomorphic graph matching and rewrite operations. The T-Core transformations required to build the symbolic execution for a DSLTrans transformation and to prove a given property need to be rebuilt for each DSLTrans transformation and each property. This step can be automatically executed using T-Core’s higher order transformation capabilities. A description of the higher order transformations required for our technique can be found in [16].

We have experimented with our symbolic execution and proof algorithms by proving properties of the Police Station transformation, described in [16]. The Police Station transformation is composed of 7 rules, including rules with at most 4 metamodel elements, indirect links and backward links. In order to test the performance of our approach we replicated the 7 rules by using different metamodel elements, up to 28 rules divided in 5 layers. The results of our experiments are shown in table 2¹. Our technique scales well up to 21 rules for our example. We stopped our experiments at 28 rules when running time became excessive, but we believe further optimizations and sophisticated encodings will increase the number of rules that can be handled.

From table 2 we can see that memory consumption is very modest, even when the symbolic execution is composed of more than one million path conditions. This is due to the fact that the actual number of DSLTrans rules used for path condition construction is very low and we only use pointers to the actual rules in memory. We can also see that the time to prove the property that holds (Prop. 1) increases with

¹The results were obtained using a 2.2 GHz Intel Core i7 machine with 8GB of DDR3 memory running Ubuntu 11.10. For each measurement involving time we repeated the given experiment three times and calculated the final result as the average of the three experiment results.

# of rules	7	14	21	28
# of path conditions	31	1051	35641	1208641
symbolic execution construction time (sec)	0.25	0.93	53.27	30513.64
used memory (Kb)	0.17	4.40	139.35	4777.00
Prop. 1, holds (sec)	0.68	6.97	320.00	-
Prop. 2, does not hold (sec)	1.8×10^{-3}	1.6×10^{-3}	1.6×10^{-3}	-

Table 2: Performance Results

the number of rules. This is due to the fact that if a property holds, then, for the time being, the whole set of path conditions needs to be explored. However, for a property that does not hold (Prop. 2) proof time is constant as proof can stop as soon as the property fails. More detailed performance results for the Police Station Transformation can be found in [16].

We are very optimistic as to the applicability of our verification approach to real world model transformations. A good indication of this fact is that by looking at several transformations we built using DSLTrans (UML to Java, Turing Machine simulation, Statecharts to Algebraic Petri Nets [6, 7, 8]), we see that the maximum number of used rules is well under 28. It is clear however that it is necessary to further experiment with variations of: the size of the rules in the considered model transformation; rule distribution among layers; number of backward links within a rule; and the number of elements of the same type scattered among different rules of the same transformation.

7 Contributions and Conclusions

In this paper we have proposed the analysis of syntactic model relation properties of model transformation via symbolic execution. We implemented our approach using model transformations written in T-Core. We have also presented early performance results. Several contributions can be identified in our work: (i) we have provided the algorithms for our original proposal of symbolic execution of model transformations in [5]. To the best of our knowledge our work provides the first attempt at explicitly building symbolic executions for a model transformation language; (ii) we show that our symbolic execution technique scales well in our experimental setting and has the potential to scale for real world problems; (iii) we demonstrate that expressiveness reduction of a model transformation language can be very beneficial to the design and construction of a model transformation verification tool; and (iv) we demonstrate that model transformations are themselves a useful tool for the proof of properties of model transformations. More generally, we provide tangible evidence that MDD principles and tools can be employed throughout the construction of MDD tools not only as mere data translators, but also at the algorithmic core of those tools. This is an indication that model transformations can indeed be used to verify model transformations.

For the future, besides exploring performance issues, we will enhance the expressiveness of our property language. Constraints on object attributes will be incorporated in the language, as will negative associations, indirect links and backward links. We are now working on applying our proof technique to model transformation properties relevant to our industrial partners, in the context of the NECSIS (Network on Engineering Complex Software Intensive Systems for Automotive Systems) project.

References

- [1] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20:42–45, Sept 2003.
- [2] T. Mens and P. Van Gorp. A Taxonomy of Model Transformations. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.
- [3] M. Amrani, L. Lúcio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J.R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *ICST*, pages 921–928. IEEE, 2012.
- [4] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *SLE*, pages 296–305. Springer, 2010.
- [5] L. Lúcio, B. Barroca, and V. Amaral. A Technique for Automatic Validation of Model Transformations. In *MoDELS*, pages 136–150. Springer, 2010.
- [6] R. Félix, B. Barroca, V. Amaral, and V. Sousa. Technical report, UNL-DI-1-2010, UNL, Portugal, 2010. <http://solar.di.fct.unl.pt/twiki5/pub/Projects/BATIC3S/ModelTransformationPapers/UML2Java.1.zip>.
- [7] DSLTrans User Manual. <http://msdl.cs.mcgill.ca/people/levi/files/DSLTransManual.pdf>.
- [8] Q. Zhang and V. Sousa. Practical Model Transformation from Secured UML Statechart into Algebraic Petri Net. Technical Report TR-LASSY-11-08, U. Luxembourg, 2011. <http://msdl.cs.mcgill.ca/people/levi/files/Statecharts2APN.pdf>.
- [9] D. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. pages 243–258. Springer, 2002.
- [10] A. Narayanan and G. Karsai. Verifying Model Transformations by Structural Correspondence. *Electronic Communications of the EASST*, 10, 2008.
- [11] F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In *ICFEM*, pages 198–213. Springer, 2012.
- [12] G. Taentzer. AGG: A Tool Environment for Algebraic Graph Transformation. In *AGTIVE*, pages 333–341. Springer, 2000.
- [13] J. De Lara and H. Vangheluwe. Atom³: A Tool for Multi-formalism and Meta-modelling. In *FASE*, pages 174–188. Springer, 2002.
- [14] D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *UML*, pages 290–304. Springer, 2004.
- [15] J.C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
- [16] L. Lúcio and H. Vangheluwe. Symbolic execution for the verification of model transformations. Technical Report SOCS-TR-2013.2, McGill U., 2013. <http://msdl.cs.mcgill.ca/people/levi/files/MTSymbExec.pdf>.
- [17] E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *ECEASST*, 29, 2010.
- [18] L. Lúcio. DSLTransVerif: A Prototype Implementation, 2013. <http://msdl.cs.mcgill.ca/people/levi/files/DSLTransVerif.zip>.