Scalable Verification of Model Transformations

Xiaoliang Wang, Adrian Rutle, and Yngve Lamo

Bergen University College (Norway)
{xwa, aru, yla}@hib.no

Abstract Model transformations are crucial in model driven engineering (MDE). Automatic execution of model transformations improves software development productivity. However, model transformations should be verified to ensure that the models produced or the transformations satisfy some expected properties. In a previous work we presented a verification approach of graph-based model transformation systems based on relational logic. The approach encodes model transformation systems as Alloy specifications which are examined by the Alloy Analyzer. But experiments showed scalability and performance problems in the approach when complex relations were present in the systems. To solve these problems, we extend our previous work by using three techniques: 1) we change the encoding to decrease the arity of relations in the derived Alloy specifications; 2) we *decompose* the expressions for the pattern matching into sub-expressions using *unique elements*; 3) we use *annotations* to decrease the complexity of the metamodel and the model transformation rules. The results of our experiments indicate that the new techniques lead to better scalability and performance.

Keywords: Model transformations; verification of model transformations; scalability problem; verification performance.

1 Introduction

In model driven engineering (MDE), models are the first class entities of the software development. They are used to specify the domain under study, to generate program code, to document software, etc. Ideally, models in a development phase can be generated automatically from models in a previous phase by model transformations. Such automation makes MDE appealing by increasing productivity. However, errors may exist in model transformations and be propagated to subsequent phases resulting in erroneous models and software. Thus, verification of model transformations is a necessary task to ensure correctness, i.e. the produced models or the transformations satisfy some expected properties.

In our previous work [22], we presented a verification approach of graph-based model transformation systems based on relational logic. A model transformation system, which consists of a metamodel and model transformation rules, is encoded automatically as an Alloy specification. Then the specification is examined by the Alloy Analyzer to verify if the system satisfies some properties within a user-defined scope. We verified conformance, safety and reachability properties

by checking the *Direct Condition* and the *Sequential Condition* [22]. However, the previous approach suffered from a scalability problem. Complex relations in metamodels and transformation rules caused relations of high arity present in the derived Alloy specifications. As a consequence, the Alloy Analyzer could not examine the specifications when using larger scopes.

To handle the scalability problem we change the encoding procedure, especially the part for matching the left and right patterns of the model transformation rules, to get rid of relations with high arity. A side effect of this change is that we get longer expressions for the pattern matching. This leads to increased verification time and worse performance. To handle the performance problem we extend our approach with two techniques: decomposition and annotation. With the first technique, we divide the expressions for the pattern matching into sub-expressions using *unique elements*. With the second technique, we use annotations to specify state information. This will decrease the complexity of the relations and the amount of the constraints in the metamodel.

Section 2 recalls the verification approach and presents some related background information. Sections 3, 4 and 5 present the changes in the encoding procedure, the usage of unique elements and the usage of annotations, respectively. In Section 6, we describe several experiments and present the results to show the effect of the new techniques. Related work is discussed in Section 7 and finally concluding remarks and future work are presented in Section 8.

2 Background

In this section, we recall the verification approach for graph-based model transformation systems detailed in [22]. Since the verification approach is based on Alloy [1] and the running example is specified in the Diagram Predicate Framework (DPF) [11,12,19,21], we give first a brief introduction to these frameworks.

2.1 Alloy

Alloy is a specification language and an analyzing tool for relational models. The language is declarative, suited for describing complex model structures and constraints based on relational logic. Artifacts in a model are defined in Alloy as *signatures* while relations among them are defined as fields of the signatures. Some predefined signatures are offered, like *Int*. Constraints on specifications are defined as *fact*s while reusable constraints with parameters are defined as *pred*s. The Alloy Analyzer is a constraint solver which verifies Alloy specifications by first translating them into SAT problems and then resolving them using a SAT solver. It can find instances which are well-typed by (and are satisfying the constraints of) the specifications using the *run* command. One can also use the *check* command to find counterexamples violating some properties. Notice that Alloy performs a bounded check, i.e. for each signature, a user-defined *scope* bounds the number of its instances.

2.2 Diagram Predicate Framework (DPF)

DPF provides formalization of (meta)modelling and model transformation based on graph transformations [13] and category theory [5]. (Meta)models in DPF are

represented by diagrammatic specifications consisting of an underlying graph and a set of constraints. Fig. 1 is the metamodel of Dijkstra mutual exclusion algorithm [10], in which P, T, R and $\{F1, F2\}$ represent process, turn, resource and the flags of a process while `[xor]` and `[inj]`, etc. are the constraints [22] Each constraint is formulated by a predicate from a predefined signature. A model is valid, i.e. the model conforms to its metamodel if there is a typing morphism from the underlying graph of the model to the underlying graph of the metamodel, and the model satisfies all the constraints of the metamodel.

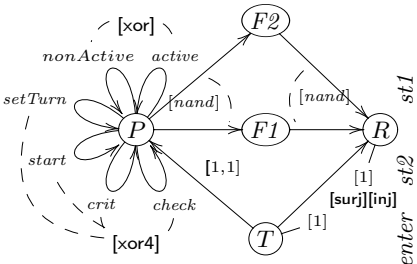


Figure 1: Metamodel

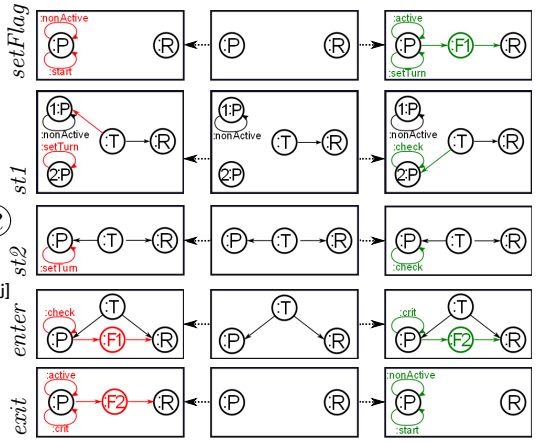


Figure 2: Transformation rules

A model transformation system consists of a metamodel (Fig. 1) together with a set of model transformation rules (Fig. 2). The transformation rules specify how a source model can be transformed to a target model. A model transformation rule $p : L \xleftarrow{l} K \xrightarrow{r} R$, consists of the left pattern L, the right pattern R and the coordination pattern K, together with two injective graph morphisms l and r. A model transformation consists of a sequence of direct model transformations (application of one model transformation rule). The execution of transformations follows the classic Double-Pushout (DPO) approach as in [13]. That is, for each match m of L in the source model S, we create a match n of R in T using two pushout constructions. DPF also provides a framework to specify constraint-aware model transformation rules [21]. That is, the transformation rules may contain constraints which may be used to control the matches and to decide what to create in the target model.

2.3 Verification Approach

In [22], as a running example we represented Dijkstra's mutual exclusion algorithm as a model transformation system (see Fig. 1 and 2). The approach uses an encoding procedure to translate the system to an Alloy specification. The metamodel and the transformation rules are encoded as corresponding signatures and predicates in the Alloy specification as follows:

1. Assume that there are m nodes and n edges in the structure of the metamodel. Each node N_i, $i \in [1..m]$ is encoded as a signature S_{N_i} while each edge E_j, $j \in [1..n]$ as a signature S_{E_j} with two fields $src : one\ E_j^s$ and $trg : one\ E_j^t$. The Alloy keyword *one* encodes that each edge has exactly one source (target) node E_j^s (E_j^t). Models are encoded as signatures S_G with two fields $nodes : set\ S_{N_1} + \cdots + S_{N_m}$ and $edges : set\ S_{E_1} + \cdots + S_{E_n}$ encoding the contained nodes and edges.
2. The DPF predicates are encoded as *pred*s in Alloy while constraints as $fact$s.
3. Direct model transformations are encoded as a signature *Trans* with 7 fields: the rule applied *rule*, the source and target model *source*, *target*, as well as, the deleted and added elements : *dns* and *ans* (nodes), *des* and *aes* (edges).
4. Each rule application is encoded as a predicate $rule_i[trans]$ stating that the transformation *trans* applies rule i. In this predicate, we encode that the source (target) model has exactly one match of the left (right) pattern in which some matched elements are deleted (added) according to the rule.
5. In order to ensure that each direct model transformation applies only one rule, for each type in the metamodel, a number restricts how many of its instances are deleted (added).

```
1 sig S_{N_i} {} //For each node N_i, i∈[1..m]
2 sig S_{E_j} {src:one E_j^s, trg:one E_j^t} //For each edge E_j, j∈[1..n]
3 sig S_G {nodes:set S_{N_1} + ... + S_{N_m}, edges:set S_{E_1}+... + S_{E_n}}
4 sig Trans{rule:one Rule, source,target:one Graph, dns,ans:set S_{N_1}+...+S_{N_m},
      des, aes:set S_{E_1}+...+S_{E_n}}
5 fact{all t:Trans | rule_1[t] or ... or rule_r[t]}
```

After applying the encoding procedure, we verify the system by using the Alloy Analyzer to examine the specification. The Alloy Analyzer searches for counterexamples by executing the command $check\{constraint\}\ for\ scope$. If no counterexample is found, the property is verified correct within the scope. Otherwise, a counterexample can be visualized to assist the designer to correct the system. In [22] we verified conformance, safety and reachability properties by checking the *Direct Condition* (i.e. every direct model transformation produces a valid target model) and the *Sequential Condition* (i.e. if the direct condition is not satisfied, for each counterexample there exists a sequence of direct model transformations that will produce a valid target model).

2.4 Challenges in Verification

The Alloy Analyzer performs a bounded check within a state space determined by the *scope*. Given an Alloy specification consisting of m signatures, a scope $[s_1, s_2, ..., s_m]$ bounds the size of the ith signature to s_i. For a relation of arity n, the size of the state space containing all the possible instances is $2^{(\sum_{i=1}^{m} s_i)^n}$ [16]. The encoding procedure, especially the encoding of the pattern matching, leads to relations of high arity in the Alloy specifications. This impairs scalability since relations of high arity hampers verification within large scopes, e.g. in [22] the highest scope was 4. In the following section, we present a new encoding for the pattern matching which reduces the arity of the relations in the Alloy specifications.

3 Changes in the Encoding Procedure

A direct model transformation is the application of one model transformation rule. As a result, the source (target) model has a match of the left (right) pattern of the rule with elements deleted (added) according to the rule. Assuming that the left (right) pattern of a transformation rule is a connected graph containing edges $e_1, \ldots, e_m$ and nodes $v_1, \ldots, v_n$ (if the pattern is a disconnected graph, each connected subgraph is encoded separately), the pattern matching is encoded as *one* e_1, ..., $e_m | p(e_1, \ldots, e_m)$, where $p(e_1, \ldots, e_m)$ is the relational expression about structure match and change. The Alloy keyword *one* enforces that only one pattern is present in the source (target) model. For example, the right pattern of the rule *setFlag* in Table 2 is encoded as follows:

```
one a:active&t.aes, s:setTurn&t.aes, pf:PF1&t.aes, f:F1R&t.aes |
 let p=a.src, f1=pf.trg, r=f.trg |
 p=a.trg and s.src=p and s.trg=p and pf.src=p and f.src=f1 and f1 in F1&t.ans
  and r in R&(t.source.nodes-t.dns) and p in P&(t.source.nodes-t.dns)
```

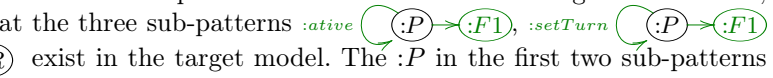
The quantification *one* e_1, ..., e_m in the relational formula causes relations of high arity in the Alloy specification. The Alloy Analyzer cannot handle such quantification with larger scopes, especially when using the keyword *one* [22]. After studying the encoding procedure, we find that the keyword *one* is not necessary because the number restriction on the deleted (added) elements implies that only one pattern is present in the source (target) model (see item 5 in Section 2.3). Therefore, we could use *some* instead of *one*. Since the expression *some* e_1, ..., $e_m | p(e_1, \ldots, e_m)$ equals to *some* $e_1 | \ldots | $ *some* $e_m | p(e_1, \ldots, e_m)$, we can split the existential quantification and evaluate the quantifiers one by one. In this way, we obtain Alloy specifications without high-arity relations. For example, the right pattern of the rule *setFlag* can now be encoded as:

```
some a:active&t.aes|let p=a.src|
 p in P&(t.source.nodes-trans.dns) and p=a.trg
  and some s:setTurn&t.aes|s.src=p and s.trg=p
  and some pf:PF1&t.aes|let f1=pf.trg|pf.src=p and f1 in F1&t.ans
  and some f:F1R&t.aes|let r=f.trg|f.src=f1 and r in R&(t.source.nodes-t.dns)
```

Thus, the scalability problem due to high-arity relations can be solved by the above mentioned change of the encoding. However, this leads to longer expressions for the pattern matching, which in turn will lead to poor performance during verification. In the following two sections, we introduce two techniques to address this problem and improve the performance. Note that in Section 6 we summarize the gains in translation time and verification time due to the optimization techniques discussed in the following sections.

4 Decomposition of Patterns

In order to get rid of long expressions for the pattern matching, we decompose them into sub-expressions during the encoding procedure using unique elements. A deleted (added) element is unique if it is the only instance of its type deleted (added) during a direct model transformation. Usually some unique elements

share a common structure. Consider the right pattern of the rule *setFlag*, in a transformation applying the rule, the added arrow :*active* in Fig. 2 is the only instance of *active* which will be added. The edges :*setTurn* and :*P*→:*F1* are unique elements sharing the same :*P*. In addition, the unique elements :*P*→:*F1* and :*F1*→:*R* share the same :*F1*. In the previous section, we encoded the pattern matching so that the whole pattern needed to exist in the target model. Here, we require that the three sub-patterns 

and 

exist in the target model. The :*P* in the first two sub-patterns are the same and the :*F*1 in the last two sub-patterns are the same because the unique elements share them. After decomposition using the unique elements, the right pattern of the rule *setFlag* is encoded as:

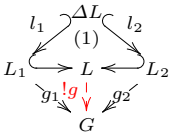
```
some a:active&t.aes|let p=a.src|p in (t.source.nodes-t.dns) and p=a.trg
 and some pf:PF1&t.aes|let f1=pf.trg|pf.src=p and f1 in t.ans
some s:setTurn&t.aes|let p=s.src|p in (t.source.nodes-t.dns) and p=s.trg
 and some pf:PF1&t.aes|let f1=PF10.trg|pf.src=p and f1 in t.ans
some e:F1R&t.aes|let f1=e.src,r=e.trg|f1 in t.ans and r in (t.source.nodes-t.
    dns)
```

Now we use the following lemma to prove that the decomposition is valid.

Lemma 1. *Assume a pattern L is divided into two sub-patterns L_1 and L_2 where $L_1 \cup L_2 = L$ and $L_1 \cap L_2 = \Delta L$ containing unique elements. In addition, the source and target nodes of the edges contained by ΔL are also contained by ΔL. We can prove that G has a match of L iff G has matches of g_1:L_1→G and g_2:L_2→G, where $g_1|_{\Delta L} = g_2|_{\Delta L}$.*

Proof. ⇒ Straightforward

⇐ l_1:ΔL→L_1, l_2:ΔL→L_2 are the two inclusion maps. Due to the assumptions, the square (1) in the right figure is a pushout. Since $g_1|_{\Delta L} = g_2|_{\Delta L}$, we can get $l_1; g_1 = l_2; g_2$. Therefore, there is a unique map $!g$:L→G.



In the new encoding procedure we identify sub-patterns sharing the same unique elements. In this way, long expressions for the pattern matching are decomposed into sub-expressions, thus gaining better translation time.

5 Annotation in Transformation Systems

In this section, we use annotations to specify state information in the metamodel and the transformation rules. The encoding procedure is changed correspondingly. The complexity of the metamodel is decreased since we reduced the amount of signatures and constraints.

5.1 Changes of Metamodel and Rules

In Section 2, arrows (e.g. *nonActive*, *start*) and nodes (e.g. *F0*) were used to specify the states of a process. In this way, complex relations appeared in the models. Furthermore, we needed to add extra constraints to correctly specify states; the metamodel in Figure 1 had 28 constraints. In order to reduce the number of the signatures and constraints, we use annotations to specify the

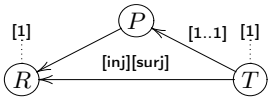


Figure 3: Metamodel

Predicate	Visualization
<Flag>	:P <F0>|<F1>|<F2>
<IsActive>	:P <active>|<nonActive>
<At>	:P <start>|<setTurn>|<check>|<crit>

Table 1: The signature Σ_2 used for annotation

Rule	L	K	R
setFlag	<nonActive> <F0> :P :R <start>	:P :R	<active> <F1> :P :R <setTurn>
st1	<nonActive> 1:P :T :R 2:P <setTurn>	<nonActive> 1:P :T :R 2:P	<nonActive> 1:P :T :R 2:P <check>
st2	:P :T :R <setTurn>	:P :T :R	:P :T :R <check>
enter	:T <check> :P :R <F1>	:T :P :R	:T <crit> :P :R <F2>
exit	<active> <F2> :P :R <crit>	:P :R	<nonActive> <F0> :P :R <start>

Table 2: Transformation Rules

state information; the new metamodel contains only 10 constraints. A similar technique was used in [20] to specify states of workflow instances.

Fig. 3 shows the metamodel using the annotation technique. The nodes R, P, T are the same as in Fig. 1. Annotations, e.g. <nonActive>, <Active> and <F1>, are used to specify state information instead of graph structures (like nodes and edges). We use three groups of annotations; for instance the group <Flag> with three annotations <F0>, <F1> and <F2>. Each group has an applicable type t in the metamodel. In an instance, each element typed by t can be marked with only one annotation from the group. In the example, all the annotation groups can be applied to instances of the type P.

Since the transformation rules in a model transformation system are defined based on the metamodel, changing the metamodel requires also changing the transformation rules. Different from the transformation rules in Section 2, the rules are now defined with constraints as shown in Table 2. This implies that we need to use constraint-aware model transformations as detailed in [21]. We use colors to denote whether an element is deleted (red), added (green) or not changed (black). The rules are almost the same as in Section 2; the difference is that the structures representing state information are replaced with annotations.

5.2 Annotation Encoding

Annotations can be encoded as normal node and edge signatures using Alloy primitive type *Int* with minor changes as follows:

```
1 sig S^E_AG{src:one N, trg:one Int}{trg>=0 and trg<n} //n is the number of
      annotations contained within the group
2 sig Graph{nodes:set S_N1+...+S_Nm+Int, edges:set S_E1+...+S_En+S^E_AG}
```

1. In each annotation group AG, each annotation is indexed with a unique number. For each AG, an edge signature S^E_{AG} is created, where its *src* field is N, the applicable type of the group; while the *trg* field is the corresponding signature index. The edge signature encodes that a node is marked with an annotation from the AG.
2. The signature *Graph* contains implicit elements Int and S^E_{AG} besides explicit elements $Node_i$ and $Edge_j$, as shown in line 2 in the listing above.

The three AG in the metamodel in Fig. 3 are encoded as follows:

```
sig AP_Flag{src:one P, trg:one Int} {trg>=0 and trg<2}
sig AP_At{src:one P, trg:one Int} {trg>=0 and trg<4}
sig AP_IsActive{src:one P, trg:one Int} {trg>=0 and trg<3}
sig Graph{nodes:set P+R+T+Int, edges:set PR+TP+TR+AP_Flag+AP_At+AP_IsActive}
```

6 Experiments and Results

In this section, we present experiments to study how the three techniques tackle the scalability problem and improve the performance. The first experiment shows how the change of encoding affects the scalability. The last two experiments show the effect of each optimization separately (see Table 3). All the experiments perform conformance verification and are executed on an Intel®Core™i5-2410M @2.30GHz*4 machine with 4GB RAM. The left column shows the scope, and each group of 3 columns shows the result of applying the techniques. Since the Alloy Analyzer uses SAT solver, we present the time cost for translation from Alloy specification to a SAT problem (TR), the verification time to solve the SAT problem (VE) and the total time (TO). Note that the table shows the total time for verifying all the constraints.

The results show that after changing the encoding procedure the approach scales better since we can verify the transformation system for larger scopes (even larger than 14). The results also indicate that the verification with the new encoding procedure is time consuming with large scopes; e.g. with scope 14 the TR is 33.8 minutes while the VE of is 8.5 minutes for all the 28 constraints.

With the decomposition technique, VE decreases to 30 seconds and the TR is around 3 minutes for the 28 constraints. The annotation technique shows even better performance: since the annotation approach reduces the constraints in the metamodel from 28 to 10, within scope 14, the TR reduces to 10125 ms, while the VE decreases to 16041ms for the 10 constraints.

Table 3: Summary of verification performances (unit:millisecond)

Scope	Old Encoding			New Encoding			Decomposition			Annotation		
	TR	VE	TO	TR	VE	TO	TR	VE	TO	TR	VE	TO
3	6235	3084	9319	4489	1615	6104	4357	1831	6188	1497	599	2096
4	9655	5236	14891	6530	1679	8209	3535	1552	5087	1786	944	2730
5				13194	2891	16085	3506	2365	5871	2320	751	3071
6				29257	5809	35066	4189	4668	8857	2357	980	3337
7				60861	9974	70835	5652	7769	13421	2703	1056	3759
8				114951	21960	136911	7173	14622	21795	2906	1544	4450
9				209371	52176	261547	10066	41465	51531	3331	2421	5752
10				357734	90784	448518	12576	45531	58107	3792	5866	9658
11				581151	162532	743683	15246	92928	108174	5272	5081	10353
12				910631	307525	1218156	19387	71090	90477	6766	9291	16057
13				1408900	307081	1715981	24797	243451	268248	8123	13673	21796
14				2030142	509732	2539874	30336	180909	211245	10125	16041	26166

Note that although the decomposition technique does not lead to as good performance as application of annotation, it is more generic in the sense that it could be applied to any model transformation system. But the annotation approach can only be used when state information is present in the system.

7 Related Work

The literature related to the verification of model transformation systems is becoming abundant. Some works propose verification approaches specialized for specific languages. For example, verification of DSLTrans [6] and ATL transformation [17] are studied in [18] and [7], respectively. Both languages are terminating and confluent by nature. Because of this, both works verify model syntax relations [2], which verify whether certain elements of the source model have been transformed into elements of the target model. Different from these works, our approach applies for verification of graph-based model transformations and properties without being restricted to a certain transformation language.

GROOVE [14] verifies graph-based model transformations using model checking. In this approach, the initial state must be given and it works only for finite state spaces. In addition, it encounters the state space explosion problem which is well-known in model checking. Simone et al. [8] uses relational structures to encode graph grammars and FOL to encode graph transformations. In this way, they provide a formal verification framework to reason about graph grammars using mathematical induction, which needs mathematical knowledge. Guerra et al. [15] proposed an automatic verification approach based on visual contract. In this approach, test input models are generated by a constraint solver and properties specified as contracts are verified by their algorithm on those input models. Similarly, our work use constraint solver techniques and is input independent [2]. By contrast, the approaches with constraint solvers are incomplete in the sense that the properties are verified within a certain coverage of instances. However, it can be used to quickly find bugs in a system and provide feedbacks about which parts of the system cause the errors. Furthermore, we do not build the whole state space. This avoids the state space explosion problem which the model checking approaches encounter. There are also several previous works [3,4,9] for verification of transformation systems with Alloy. But they only give encoding of specific examples, without offering a general encoding procedure.

8 Conclusion and Future Work

In this paper we have presented an extension of our previous work [22] to tackle scalability and performance issues by employing three techniques: changing the encoding procedure, decomposing patterns into sub-patterns and applying annotation to represent state information. The experimental results show that the techniques improve the scalability and performance (i.e. translation and verification time). In the future we will examine how the approach can be used to verify other properties like liveness, deadlock, etc. Furthermore, more cases will be examined to study the effects of the optimization techniques and evaluate how the approach performs for more complex model transformation systems. Currently, we encode direct model transformations applying only one rule, but in future we would like to encode model transformations which apply rules concurrently (composition of direct model transformations).

References

1. Alloy. *Project Web Site*. `http://alloy.mit.edu/community/`.
2. M. Amrani, L. Lúcio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. R. Cordy. J.r.: A tridimensional approach for studying the formal verification of model transformations. In *Proc. of VOLT*, 2012.
3. K. Anastasakis, B. Bordbar, and J. M. Küster. Analysis of Model Transformations via Alloy. In *Proc. of MoDeVVa*, 2007.
4. L. Baresi and P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *Proc. of ICGT*, 2006.
5. M. Barr and C. Wells. *Category Theory for Computing Science (2^{nd} Edition)*. Prentice-Hall, Inc., 1995.
6. B. Barroca, L. Lucio, V. Amaral, R. Félix, and V. Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Proc. of SLE*, 2010.
7. F. Büttner, M. Egea, and J. Cabot. On verifying ATL transformations using off-the-shelf SMT solvers. In *ACM/IEEE MODELS 2012*, LNCS, 2012.
8. S. A. da Costa and L. Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming*, 77:480–504, 2012.
9. Z. Demirezen, M. Mernik, J. Gray, and B. Bryant. Verification of DSMLs Using Graph Transformation: A Case Study with Alloy. In *Proc. of MoDeVVa*, 2009.
10. E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 1965.
11. Z. Diskin. *Encyclopedia of Database Technologies and Applications*, chapter Mathematics of Generic Specifications for Model Management I and II. Information Science Reference, 2005.
12. Z. Diskin, B. Kadish, F. Piessens, and M. Johnson. Universal Arrow Foundations for Visual Modeling. In *Diagrams 2000: 1^{st} International Conference on Diagrammatic Representation and Inference*, 2000.
13. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer. Springer-Verlag New York, Inc., 2006.
14. A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 2012.
15. 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), 2013.
16. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
17. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2), 2008.
18. J. D. S. M. Levi Lúcio and H. Vangheluwe. A technique for symbolically verifying properties of graph-based model transformations. Technical Report SCOCS-TR-2012, McGill University, 2013.
19. A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, 2010.
20. A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Proc. of BM-FA*, 2012.
21. A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAP*, 2012.
22. X. Wang, Y. Lamo, and F. Büttner. Verification of graph-based model transformation using Alloy. In *Proc. of GTVMT*, 2014.