Electronic Communications of the EASST
Volume 67 (2014)

Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014)

Verification of Graph-based Model Transformations Using Alloy

Xiaoliang Wang, Fabian Büttner, Yngve Lamo

13 pages

Guest Editors: Frank Hermann, Stefan Sauer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verification of Graph-based Model Transformations Using Alloy

Xiaoliang Wang[1], **Fabian Büttner**[2], **Yngve Lamo**[1]

[1] xwa@hib.no,yla@hib.no
Bergen University College, Norway

[2] fabian.buettner@gmx.org
École des Mines de Nantes-INRIA, France

Abstract: Model transformations are fundamental in model driven development. Thus, verification of model transformations is indispensable to ensure the quality and the reliability of transformation results. In this paper we focus on graph-based model transformation systems using the double-pushout (DPO) approach and study their correctness w.r.t. conformance. It means that, given a transformation system and a valid source model, any applicable sequences of model transformations will produce a valid target model. A procedure is presented to verify firstly if a model transformation system is correct w.r.t. conformance by checking the *Direct Condition*, i.e., each direct model transformation produces a valid target model from a valid source model. Then, for systems not satisfying the direct condition, it checks the *Sequential Condition*, i.e., if a direct model transformation t produces an invalid target model from a valid source model, then there exists a sequence of direct model transformations succeeding the transformation t that produces a valid target model. The satisfiability of the latter condition cannot always promise correctness, but it ensures that, from every valid source model, a valid target model *can* be produced. The procedure uses a bounded verification approach based on First Order Logic (FOL). The approach encodes a transformation system and the two conditions into a relational logic specification in Alloy. Then the specification is inspected by the Alloy Analyzer to check if the system satisfies the conditions. When it violates the conditions, the analyzer presents a counterexample, that may be used to redesign the system. An example is given to illustrate the bounded verification approach in the Diagram Predicate Framework (DPF).

Keywords: Model transformation system; Conformance; Alloy; Verification

1 Introduction

In model driven engineering (MDE), models are the basis for software development. They are used to specify the domain under study, to generate program code and for documentation purposes etc. Ideally, a model in the next development phase can be automatically generated from a model in the previous phase by model transformations. Such automation makes MDE appealing by producing software with better quality at higher productivity. However, validation of model transformations should be ensured. Without validation, errors in some transformations are propagated to subsequent phases, which may cause erroneous software at the end.

In this paper, we focus on graph-based model transformations [EEPT06]. Such model transformations are usually executed by applying model transformation rules on models. The rules tell how to execute a transformation and generate a target model from a source model. A model transformation system consists of a metamodel and a set of such rules along with a mechanism to control rule applications. Our work aims to verify if a model transformation system is correct w.r.t. conformance. Besides, for systems accepting invalid intermediate models in a sequence of model transformations, a weaker correctness condition is verified.

We will present a procedure to verify firstly if a model transformation system is correct w.r.t. conformance by checking if it satisfies a *Direct Condition*. Then, for systems not satisfying this *strong* condition, a weaker *Sequential Condition* is checked. The procedure utilizes a bounded verification approach based on First Order Logic (FOL). The main idea is to encode automatically a model transformation system as a relational logic specification in Alloy [All], a modelling language based on FOL. Then we resort to the Alloy Analyzer [All], which can generate instances or counterexamples of a model. To verify if the system satisfies the conditions the analyzer checks if the specification has any counterexamples within a user-defined bounded scope.

To illustrate our approach, we present a running example in Diagram Predicate Framework (DPF) [RRLW09]. DPF provides diagrammatic modelling of both structure and constraints based on category theory. Besides, it also offers functionality to specify graph-based model transformations. We will present a DPF model transformation specification and show how it can be encoded as an Alloy specification and verified with the Alloy Analyzer.

The paper is organized as follows. Section 2 firstly shows how to specify a model transformation system in DPF. Then Section 3 explains the verification procedure and the FOL-based bounded verification approach. In Section 4 we give a short introduction to Alloy, explain the encoding process from model transformations to relational logic specifications. Afterwards, in Section 5 we present the result of the check of the *Direct Condition* and the *Sequential Condition*, before we discuss the given approach. Section 6 compares our work with related research and finally Section 7 concludes the paper and envisions some future research directions.

2 Model Transformation System

In this section, we will show how to specify a model transformation system in DPF. A model transformation system [EEPT06] consists of a metamodel **M** and a set of model transformation rules. Figure 1 and 2 show a variant of Dijkstra's algorithm for mutual exclusion [Dij01] as a model transformation system in DPF. The algorithm ensures that a critical resource is exclusively accessed by one process each time. In DPF, the structural syntax of a model is represented by a directed graph and constraints as diagrammatic predicates on part of the graph. Those predicates are denoted by $[PredicateName]$. Note that each predicate has a specific shape graph defining which kind of graphs the predicate can constrain. For more information about predicate in DPF, refer to [YXF+13]. Figure 1 is the metamodel used in the example. Because of space limitation, some constraints are not shown in the example. R is the resource which processes P can access. The node T tells which process that can access the resource. An arrow $T \rightarrow P$ means that the process P is eligible to access the resource R. The flags $\{F1, F2\}$ and the states $\{nonActive, active, start, crit, check, setTurn\}$ are used to control access to the resource. An arrow

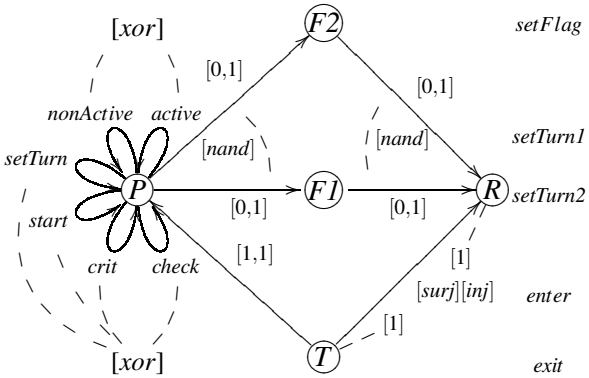

Figure 1: Metamodel

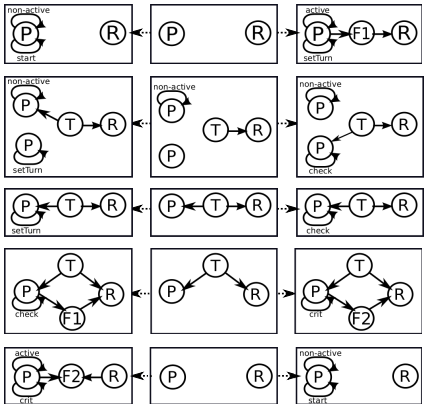

Figure 2: Transformation rules

from a process to one of the two flags means that the process is tagged with such a flag. A reflexive arrow on a process P labeled with one of the six states means that the process is in such a state. Moreover, the model needs to satisfy the following constraints:

1. There is exactly one resource in a model, which is ensured by the multiplicity $[1]$ on R. A similar constraint also applies to T.
2. The states $nonActive, active, start, crit, check, setTurn$ should be $[reflexive]$.
3. Every process should be in one of the states *nonActive*, *active*, and also in one of the states *start*, *crit*, *check*, *setTurn*. This is ensured by the $[xor]$ constraints between the *nonActive* and the *active* states, and among the *start*, *crit*, *check*, *setTurn* states.
4. Every process should only be in one of the state combinations $\{nonActive, start\}$, $\{active, crit\}$, $\{active, check\}$ or $\{active, setTurn\}$.
5. Every process may be tagged with at most one of the flags $\{F1, F2\}$. This is ensured by the multiplicity constraints $[0,1]$ on both arrows $P \to F1$ and $P \to F2$, along with a $[nand]$ constraint between the two arrows.
6. Each flag may have at most one arrow to R, which is ensured by the multiplicity constraint $[0,1]$ on the respective arrows and a $[nand]$ between the arrows.
7. $T \to R$ is bijective, this is ensured by $[inj]$ and $[surj]$ constraints.
8. It is at most one process in the system that is in state *crit*, is marked with the *F2* flag and is eligible to access the resource. This constraint is ensured by $[critical]$.

A metamodel M, including structural syntax and constraints, defines its valid instances which conform to M. The conformance states that an instance I should be typed by M, denoted as $I : M$. Formally, it means that there is a graph morphism from the graph of I to the graph of M. Moreover, it also states that I satisfies all the constraints defined on M, denoted as $I \rhd M$. The following figures show several graphs. Names of instance elements are omitted for brevity. For example, $\textcircled{P}$ denotes $p1 : P$, an instance element $p1$ typed by P. Figure 3 is a valid instance of Figure 1 since it is well-typed and satisfies all the constraints; the others are not because $P \to T$

causes type violation in Figure 4; in Figure 5 P is not in state *active*, which violates constraint 4.

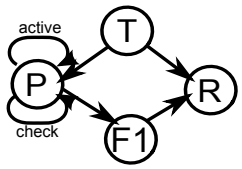

Figure 3: An Instance

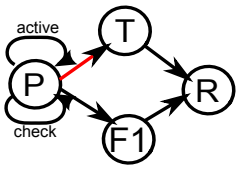

Figure 4: An invalid model

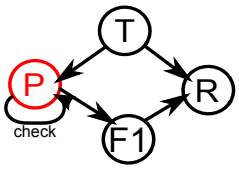

Figure 5: An invalid model

Based on the modelling framework, DPF also provides a framework to specify constraint-aware model transformations[RRLW12], which means that the transformation rules may contain constraints. However, in this paper, we only consider metamodel constraints. In a model transformation rule $\{p : L \xleftarrow{l} K \xrightarrow{r} R\}$, L, K and R are the left-hand side, the gluing graph and the right-hand side, while the two graph morphisms l and r are injective. L and R are typed by $\mathbf{M}$, but not necessary valid instances of $\mathbf{M}$. The model transformation rules in the example are shown in Figure 2. Rule *setFlag* requests access to the resource. Rule *setTurn1* and *setTurn2* assign T to one process depending on the context. Rule *enter* enables the eligible process to access the resource, while rule *exit* releases the resource after access.

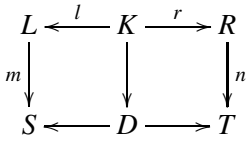

Figure 6: Double-pushout Diagram

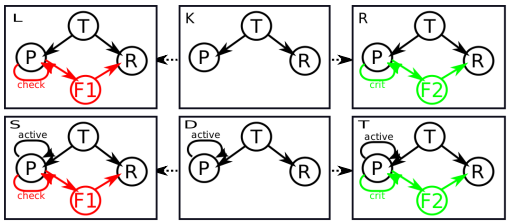

Figure 7: A model transformation

In this paper, the execution of model transformations follows the DPO approach [EEPT06]. For each transformation rule p, given an instance S, if there is a graph morphism $m : L \to S$, we say that S has a match m of rule p. A direct model transformation is executed according to the morphisms l and r and the match m by completing the double-pushout diagram in Figure 6. After the direct model transformation, a target model T is produced. For example, the rule *enter* changes the flag form *check* to *crit*, and the state of a process from *F1* into *F2*. In the direct model transformation in Figure 7, edge *check* and node *F1* are deleted while edge *crit* and node *F2* are added. Other elements are preserved in the target model.

3 Bounded Verification of Graph-Based Model Transformation

In this section we will present a procedure to verify graph-based model transformation systems, using a FOL-based bounded verification approach. The procedure firstly verifies if a system is correct w.r.t conformance, i.e., given a valid source model, every sequence of direct model transformations from the model can produce a valid target model. This is performed by check-

ing if the system satisfy the *Direct Condition*, i.e., every direct model transformation is valid. A direct model transformation is valid if it produces a valid target model from a valid source model. The *Direct Condition* is quite strong in the sense that, in a sequence of direct model transformation, no invalid intermediate model is allowed. However, some systems accept such intermediate models and only require that the final target model should be valid. In such a situation, we weaken the correctness condition by checking the *Sequential Condition*, i.e., for each counterexample $S \to T_0$, a sequence of direct model transformations $T_0 \to \cdots \to T_n$ can produce a valid target model T_n. If the condition is satisfied, we assure that, given any valid source model, a valid target model *can* be produced after some sequences of direct model transformations. It is weaker since it does not promise to produce a valid target model after every sequence of direct model transformations.

In this procedure, a bounded verification approach based on FOL is used to check if a graph-based model transformation satisfies the *Direct Condition* or the *Sequential Condition*. The idea is to encode a model transformation system into a relational logic specification. The encoding can be executed automatically. Each component, metamodel (including structure and constraints) and model transformation rules, can be encoded in relational logic. The structure of a metamodel is encoded as functions and predicates, representing all the possible model instances typed by the graph. Besides, the semantics of model transformations, or how to execute a transformation, is also considered in the approach. In each direct model transformation, according to the applied rule and the DPO approach, some elements in the source model are deleted while some elements in the target model are added. Except those elements the rest of the source model is preserved in the target model. In this way, a direct model transformation can be encoded as the following two functions $add : T \to S$ and $delete : S \to T$:

$$add(e) = \begin{cases} NULL & \text{if e is added to T} \\ e & \text{otherwise} \end{cases} \qquad delete(e) = \begin{cases} NULL & \text{if e is deleted from S} \\ e & \text{otherwise} \end{cases}$$

Based on the direct model transformation encoding, a graph-based model transformation system, the *Direct Condition*(1) and the *Sequential Condition*(2) can be expressed as the following FOL expressions ($\mathbf{S} \to \mathbf{T}$ denotes that $\mathbf{S}$ is transformed into $\mathbf{T}$, while $\Rightarrow$ is implication in FOL):

$$\forall \mathbf{S}, \mathbf{T} \quad : \quad \mathbf{S} \rhd \mathbf{M} \wedge (\mathbf{S} \to \mathbf{T}) \Rightarrow \mathbf{T} \rhd \mathbf{M} \tag{1}$$

$$\forall \mathbf{S}, \mathbf{T_0} \quad : \quad (\mathbf{S} \rhd \mathbf{M} \wedge \mathbf{S} \to \mathbf{T_0} \wedge \neg(\mathbf{T_0} \rhd \mathbf{M})) \Rightarrow \tag{2}$$

$$\exists \mathbf{T_1}, ..., \mathbf{T_n} : \bigwedge_{i=1}^{n-1} \neg(\mathbf{T_i} \rhd \mathbf{M}) \wedge \mathbf{T_n} \rhd \mathbf{M} \wedge \bigwedge_{i=0}^{n-1} \mathbf{T_i} \to \mathbf{T_{i+1}}$$

4 Encoding of Graph-based Model Transformation Systems

In this section, we will give the details about how to encode model transformation systems as a relational logic specification in Alloy. Before that, a brief introduction to Alloy will be given. Alloy consists of the Alloy specification language, used to define specifications or models, and the Alloy analyser, used to reason about specifications. It is developed at MIT by a team led by Daniel Jackson [All]. The Alloy specification language is a declarative language suited to describe complex model structures and constraints in a simple structural modelling language based

on relational logic. The Alloy language defines a model structure as *signatures*. Each *signature* defines a typed element in the structure, representing a set of instances of this type. Relations among the typed elements are defined by the fields of the signatures. Constraints on the structure can be defined as *facts* while predicates as *pred*. The Alloy analyser can find valid instances well-typed by the structure and satisfying its constraints by executing the *run* command. It can also verify some properties by calling the *check* command to find counterexamples. Notice that Alloy performs a bounded check, i.e., for each signature, a user-defined scope bounds the number of its instances. The Alloy Analyser performs the verification within a scope by running the *command* $\{constraint\}$ *for scope*. For example, $run\{\}$ *for m* is used to find a valid instance of a model specification within a scope containing at most m instances of each signature. Assuming a specification contains n signatures, its instances contains at most $m*n$ elements. Here we only mention the basic functionalities of Alloy, please refer to [All] for more information.

4.1 Encoding of Metamodels

```
sig Graph{                                    sig Node_i{}//1 ≤i≤m
    nodes:set Node_1+...+Node_m,              sig Edge_j{//1≤j≤n
    edges:set Edge_1+...+Edge_n                  src:one Node_s,//1≤s≤m
}                                                trg:one Node_t}//1≤t≤m
fact{all g:Graph|all e:Edge_j&g.edges|g.src in g.nodes and g.trg in g.
   nodes}//1≤j≤n
```

In graph-based model transformation systems, metamodel structures are type graphs. A type graph, containing m nodes and n edges, is encoded as a *Graph* signature with set fields *nodes* and *edges*, representing all the nodes and edges. $nodes : set\ Node_1 + \cdots + Node_m$ means that each element of *nodes* can be typed by any node in the graph, where $+$ is set union in Alloy. This also applies to the field *edges*. Each node is encoded as a *Node* signature without any field while each edge as an *Edge* signature with two fields *src* and *trg* of corresponding node types. Both fields are declared with Alloy multiplicity constraint *one*. This means that each edge have exactly one source node and one target node. Besides, the *fact* constrains that every edge and its source and target nodes should be in the same graph (*in* is subset relation in Alloy).

Besides structural information metamodels contain also constraints. Constraints in DPF workbench are specified in Java or OCL [LWM+12]. They are encoded as FOL expressions in Alloy. Furthermore, the *Direct Condition* requires the precondition that the source model S is valid in every direct model transformation, while the *Sequential Condition* requires the precondition that in each verification step, for each sequence of direct model transformations $S \to T_0 \cdots \to T_k$, the source model S is valid while the intermediate model T_i is invalid for $i \in (0,k)$. For convenience, a predicate $pred\ valid[graph : Graph]$ is used to tell if a model satisfies some constraints. The constraint 5 in the example is encoded as follows:

```
pred valid[graph:Graph]{
all n:(P&graph.nodes)|lone e:PF2&graph.edges|e.src=n
all n:(P&graph.nodes)|lone e:PF1&graph.edges|e.src=n
all n:(P&graph.nodes)|not ((some e:PF1&graph.edges|e.src=n) and (some
    e:PF2&graph.edges|e.src=n))}
```

4.2 Encoding of Direct Model Transformations

```
1 sig Trans{rule:one Rule,source,target:one Graph, dnodes, anodes:set
    Node_1+...+Node_m,dedges, aedges:set Edge_1+...+Edge_n}
2 all e:Edge|((e.src in t.dnodes or e.trg in t.dnodes) implies edge in
    t.dedges) and ((e.src in t.anodes or e.trg in t.anodes) implies
    edge in t.aedges)
3 t.dnodes in t.source.nodes and t.dedges in t.source.edges
4 t.anodes in t.target.nodes and t.aedges in t.target.edges
5 t.source.nodes-t.dnodes=t.target.nodes-t.anodes
6 t.source.edges-t.dedges=t.target.edges-t.aedges
7 rule_1[t] or ... or rule_n[t]
```

Given a graph-based model transformation system, a direct model transformation $trans : S \xrightarrow{p} T$ is executed by applying a model transformation rule $p = L \xleftarrow{l} K \xrightarrow{r} R$ with the DPO approach. A direct model transformation is encoded as a *Trans* signature. In the signature, the *rule* field denotes the rule applied in a transformation, while the *source* and *target* fields represent the source and the target models. The other four fields represent the deleted elements and added elements. Besides, a transformation *trans* satisfies some constraints: there is no dangling edge after the transformation; the deleted elements and the added elements should be subset of the source graph's and the target graph's elements, respectively; the source graph and the target graph should be the same except for the deleted and the added elements. $trans.dnodes$ and $trans.dedges$ denote the deleted elements in the transformation, while $trans.anodes$ and $trans.aedges$ the added elements. $trans.source.nodes - trans.dnodes$ represents the preserved nodes in the source model while $trans.target.nodes - trans.anodes$ the preserved nodes in the target model. According to the semantics of transformations, those two sets of preserved nodes are equal. A similar constraint applies also to arrows. These are encoded as the constraints on lines 5-6.

Besides the general properties of a transformation, for each rule p, a predicate $pred\ rule_p[trans]$ encodes if a rule p is applied to a transformation. Without loss of generality, we enforce that a direct model transformation applies only one rule each time.

1. There is only one match m from L to S. During the transformation, the part $m(L) \setminus m(l(K))$ of S is deleted. Similarly, there is also only one "match" n from R to T and the part $n(R) \setminus n(r(K))$ of T is added.

   ```
   one m:L→S|(all n:m(l(K_N))|n in t.source.nodes-t.dnodes) and (all
       e:m(l(K_E))|e in t.source.edges-t.dedges)and (all n:m(L_N)\m(l(K_N))
       |n in t.dnodes) and (all e:m(L_E)\m(l(K_E))|e in t.dedges)
   one n:R→T|(all n:n(r(K_N))|n in t.target.nodes-t.anodes) and (all
       e:n(r(K_E))|e in t.target.edges-t.aedges) and (all n:n(R_N)\n(r(K_N))
       |n in t.anodes) and (all e:n(R_E)\n(r(K_E))|e in t.aedges)
   ```

2. During a transformation, no element in S is deleted except the elements matched by $L \setminus l(K)$. To fulfill this, a constraint restricts the number of deleted elements. For each type t having instances in $L \setminus l(K)$, the number of deleted elements typed by t in S equals to N_t, the number of elements typed by t in $L \setminus l(K)$. A similar constraint applies to the added

elements. In Alloy, # calculates the cardinality of a set. *trans.source.nodes&t* contains elements in source typed by t where & is set disjunction in Alloy. For example, only one edge of type TP is deleted in rule *setTurn*0, encoded as the constraint on line 3.

```
1 #{n:t.source.nodes&t|n in t.dnodes}=Nt
2 #{e:t.source.edges&t|e in t.dedges}=Ne
3 #e:t.dedges&TP=1
```

3. All the elements in S typed by t having no instance in $L \setminus l(K)$ are unchanged. Similarly, all the elements in T typed by t having no instance in $R \setminus r(K)$ are unchanged. For example, no node and edge typed by P and $T \to P$ is deleted or added when applying rule *setTurn*2 to a transformation. The constraint is encoded as:

```
no n:trans.dnodes&P                no n:trans.anodes&P
no e:trans.dedges&TP               no e:trans.aedges&TP
```

The encoding of rule *setTurn2* to a predicate is shown follows (part of the predicate is omit):

```
1 pred rule_setTurn2[t:Trans]{
2     some t.rule&setTurn2
3     one se_tp0:TP&(t.source.arrows-t.darrows),se_st0:setTurn&t.
          darrows, se_tr0:TR&(t.source.arrows-t.darrows)|let sv_t0=
          se_TP0.src,sv_t0=se_TP0.trg,sv_r0=se_TR0.trg|(sv_p0=se_st0.src
           and sv_p0=se_st0.trg and sv_t0=se_tr0.src and sv_p0 in P&(t.
          source.nodes-t.dnodes) and sv_t0 in T&(t.source.nodes-t.dnodes
          ) and sv_t0 in R&(t.source.nodes-t.dnodes))
4     ...
5     #setTurn&t.darrows=1 #check&t.aarrows=1
6     no P&t.dnodes ... no R&t.dnodes
7     no P&t.anodes ... no R&t.anodes
8     no check&t.darrows ... no F2R&t.darrows
9     no crit&t.aarrows ... no F2R&t.aarrows}
```

5 Result of Verification

After encoding a graph-based model transformation system to an Alloy specification, the Alloy Analyzer is used to verify the system by finding counterexamples w.r.t. different conditions.

5.1 Check Direct Condition

When checking the direct condition, the Alloy Analyzer examines if some direct model transformations produce an invalid target model from a valid source model. For example, we use the following command to find a counterexample violating constraint $[surj]$ on $PF1 : P \to F1$:

```
1 check{all trans:Trans}|all n:trans.target.nodes&F1|one e:trans.target
      .edges&PF1|e.trg=n}for 3 but exactly 1 Trans, exactly 1 Graph,
     exactly 1 Rule, exactly 1 R, exactly 1 T, exactly 1 TR
```

In the command, $fo3$ *but* ..., *exactly* 1 *TR* is the scope. It restricts the counterexample containing at most 3 instances of each signature. The scope is further restricted: since only direct model transformations are examined when checking the direct condition, for all systems, we enforce that there are exactly 1 Transformation, 2 Graphs and 1 Rule. Besides, we ensures exactly 1 R, 1 T and 1 *TR* for this example. The Alloy keyword *exactly* implements these constraints.

If a counterexample is found violating a *constraint*, the Alloy Analyzer visualizes it showing which rule is applied. We call such a rule *a problem rule* and say that the rule violates the *constraint*. The designer can utilize the information to fix the problem, e.g., adding more constraints to the metamodel or consider if some NACs are necessary for the problem rules. If no counterexample is found for any constraint, the system is verified correct within the scope.

For the example, the Alloy Analyzer gives counterexamples when checking constraints 3 and 5 (multiplicity on $P \to F1$ and $[nand]$ between $P \to F1$ and $P \to F2$). Part of a counterexample violating constraints 3 is given in Fig 8. Nodes with $ are elements violating the constraint. After

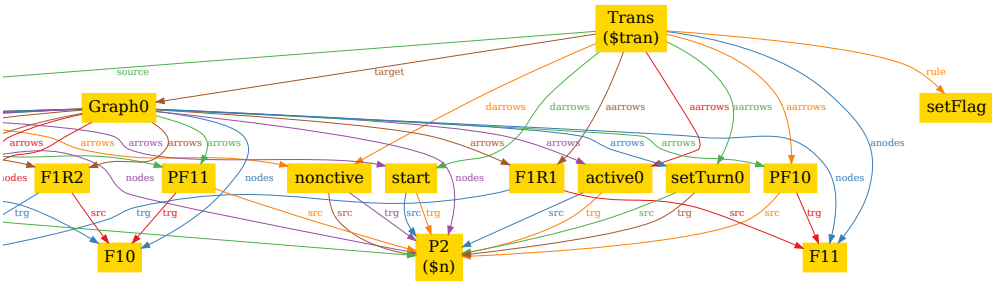

Figure 8: A counterexample

analyzing the counterexample, we find that the following constraints are missing. After adding them, the Alloy Analyzer gives no further counterexamples, which indicates that the system is correct w.r.t. conformance.

```
A process has no flag iff it is in {nonActive, start} state
A process has F1 flag iff it is in {active,setTurn} or {active,check} state
A process has F2 flag iff it is in {active,crit} state
```

5.2 Check Sequential Condition

We also apply our approach to a revised version of an example from [LMRL13] which transforms Object Oriented systems (OO) into entity-relationship (ER) models. The authors use rule amalgamation mechanism to perform transformations in their work; we verify the system using DPO approach without amalgamation. Note that in this example, the source metamodel and the target metamodel are different. Before applying the approach to such systems, we construct a joint metamodel containing the two metamodels and correspondences between elements. Several constraints are considered to verify the system. For example, (c_1) each class should be transformed into a table; (c_2) each table should have a primary key. Because of space consideration, we only briefly present the result here. When checking the *Direct Condition*, we found that both constraints are violated by some rules. After using the counterexamples to fix the system, c_2 is

still violated by a rule. In the following, we show how to check the *Sequential Condition*. For c_2, we first consider a path consisting of 2 direct model transformations in a sequence. The source model, $trans_0.source$, is valid, as shown on line 2; the intermediate model, $trans_1.target$, violates c_2. For each model violating a constraint c, $violation[c]$ generalize how the model violates the constraint, while $fix[v]$ generalizes the solution to fix the violation. For example, a violation for c_2 is apparently that, some table has no primary key. While for the violation, the solution is to add a primary key for the table. Removing the table is not considered since no deletion happens in the example. The violation and the correction are presented below on line 3:

```
1 sig Path{t0,t1:Trans}{t0.target=t1.source}
2 all tb:Table&t0.source.nodes|some pk:PK&t0.source.edges|pk.src=tb
3 some tb:Table&t0.target.nodes|(no pk:PK&t0.target.edges|pk.src=tb) and
    (some pk1:PK}&t1.aedges|pk1.src=tb)
```

Before checking the *Sequential Condition*, it should be assured that there exists a rule fixing such a violation. This is performed by executing command $run\{\}$ *but exactly* 1 *Path*, *exactly* 2 *Trans*, *exactly* 3 *Graph*, *exactly* 2 *Rule* to see if such a path exists. If no instance is given, we can conclude that the system is not correct, i.e., from some valid source models, no model transformations can produce valid target models. Otherwise, we continues to check the condition. The check is enforced by running the command *check* $\{all\ path : Path | Table\_one\_pk[path.trans2.target]\}$ with the same scope. If no counterexample is present, it is assured that each violation of a constraints can be fixed by some succeeding transformations. Therefore, we verify that the *Sequential Condition* is satisfied. Otherwise, a longer path should be checked. In the example, the *Sequential Condition* is satisfied with a path of length 2.

5.3 Discussion

In order to show the performance of our approach, the verification times of checking the *Direct Condition* for the example presented in Section 2, is shown in Table 1. All the cases are preformed in the Alloy Analyzer on a Windows machine with a Intel Core i5-2410M processor and 4GB memory. The second line is for the scope for 3 *but* ... (s_3), while the third line is for the scope for 4 *but* ... (s_4). Each column represents the time needed to verify a specific constraint. It shows that the constraints are verified in a reasonable time. The longest time is 2700ms and 2371ms when checking $[xor]$ between the 4 states with the scope s_4. Because of the capacity of the Alloy Analyzer, the verification cannot be performed with the scope for 5 *but* ....

Table 1: Verification times(ms)

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
s_3	133	62	105	49	24	40	47	98	24	601	495	100	67	73	24
s_4	258	136	418	126	56	171	131	442	80	2700	2321	489	204	289	144

We have shown that the approach helps to find defects in graph-based model transformations. But several limitations should be noticed. The Alloy Analyzer performs a bounded check within a state space determined by a user-defined scope. Given a 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 $2\hat{}(\sum_{i=1}^{m} s_i)^n$. The analyzer searches exhaustively within the state space for instances or counterexamples of a specification. Alloy uses some optimisation to decrease the state space, but the state space still grows super-exponentially with the scope. In the approach, the transformations encoding introduces some relations with high arity. For example, rule *enter* is encoded as a relation with arity 5 (The largest number of the edges in the left and the right side). As a result, the approach will not scale well when applying to large systems or complex transformation rules.

However, some optimisation can be deployed to apply the approach to larger systems. We verify systems by finding counterexamples violating a certain constraint. When a constraint is checked, some elements in the metamodels and rules are not affected. They could be removed from the scope. For example, in Figure 2, arrow $T \rightarrow R$ is not added or deleted by any rules. When checking constraint 3 and 5, the arrow cannot affect the verification result. Those unrelated elements could be removed during verification. This could also be used to simplify a counterexample, when it is too complex to analyse. *PF2* and *crit* are not deleted or added by any transformation applying rule *setFlag* and they do not affect the satisfiability of the constraint. They are unrelated elements and could be removed from the scope when verifying the constraints. A further restricted scope with 0 *PF2*, 0 *crit*, can simplify the counterexample.

6 Related Work

Different verification techniques have been applied to model transformation verification. The tool, GROOVE [GdR+12], verifies graph-based model transformation using model checking. In this approach, the initial state must be given and it works only for finite state spaces. Besides, it encounters the state space explosion problem similar to other model checking approaches. Basil Becker et al. [BBG+06] encode safety properties as inductive invariants and present algorithms to verify safety properties using model checking. The approach is largely dependent on the number and especially the complexity of the rules and invariants. Similar to our approach no start state is required. In contrast, we also offer a procedure to verify properties along a sequence of model transformations.

Another verification technique, theorem proving, is also used in this area. Simone et al. [CR12] 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. Similarly, Leila et al. [RDCD10] translate graph grammars into Event-B specifications and use theorem provers available for Event-B to analyze the reachable states. However, the approaches is not automatic and requires mathematical knowledge. In contrast, our approach offers an automatic procedure to verify the validity of produced models.

Automatic verification of model transformations is an evolving area of research; Several methods have already been proposed. Troya and Vallecillo [TV11] provide a rewriting logic semantics for ATL [JABK08] and use Maude to simulate and verify transformations; Büttner et al. [BEC12] provide a first-order logic encoding for ATL transformations and employ SMT solvers to check their correctness. There are furthermore several verification approaches that use OCL constraints to capture or specify semantics of rule-based transformations (sometimes called transformation models), employing existing OCL model finders to check correctness prop-

erties [GLW+13, CCGL10]. Those approaches are specific to some transformation languages. While in our approach, we consider general cases to systems which conforms to the semantics of model transformation system.

The works of Anastasakis et al. [ABK07] and Baresi and Spoletini [BS06] are closest to our contribution, as they also use Alloy to analyze graph-based model transformation. They demonstrate, by example, the feasibility of representing such a model transformation in Alloy. On the other hand, they specify a (bounded) sequence of transformation rules in Alloy to check properties related to a sequence of model transformations, e.g., if a model can be produces after a sequence of model transformations. By contrast, we provides an automatic encoding of graph-based transformation systems to Alloy specifications. Besides, we address the checking of individual transformations: We verify that all direct model transformations produce a valid target model from a valid source model, which implies that the system is correct. Furthermore, we also verify if, from every valid source model, a valid target model can be produced.

7 Conclusion and Future Work

Model transformations are of great importance in MDE, hence the correctness of model transformations should be ensured. We proposed an approach to verify static correctness (preservation of metamodel conformance) of graph-based model transformations. The verification is realized by automatically encoding systems into relational logic formulas which can be checked by Alloy (given a bounded search space, using a SAT solver). At the current stage, the examples show that our verification approach can check individual transformations and verify systems allowing intermediate instances in a sequences of transformations. But the approach is not restricted to the verification w.r.t. conformance. It also could be extended to verify safety proprieties and liveness proprieties. We will systematically study when this approach should be used. Besides, much more work should established in the future, e.g., how to automatically derive the unrelated elements when verifying a constraint; Currently, the counterexamples are visualized with Alloy tools. How to present the result diagrammatically in DPF should be studied for a more user-friendly feedback. Furthermore, the scalability problem should be addressed in the future.

Bibliography

[ABK07] K. Anastasakis, B. Bordbar, J. M. Küster. Analysis of Model Transformations via Alloy. In *MoDeVVa'2007, Proceedings*. 2007.

[All] Alloy. Project Web Site. http://alloy.mit.edu/community/.

[BBG+06] B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling. Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06, pp. 72–81. 2006.

[BEC12] F. Büttner, M. Egea, J. Cabot. On verifying ATL transformations using off-the-shelf SMT solvers. In *ACM/IEEE MODELS 2012*. LNCS. 2012.

[BS06] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *ICGT*. LNCS 4178. 2006.

[CCGL10] J. Cabot, R. Clarisó, E. Guerra, J. de Lara. Verification and validation of declarative model-to-model transformations through invariants. *JSS* 83(2), 2010.

[CR12] S. A. da Costa, L. Ribeiro. Verification of graph grammars using a logical approach. *Science of Computer Programming* 77:480–504, 2012.

[Dij01] E. W. Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*. 2001.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer. Springer-Verlag New York, Inc., 2006.

[GdR+12] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer* 14:15–40, 2012.

[GLW+13] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger. Automated Verification of Model Transformations Based on Visual Contracts. *Autom. Softw. Eng.* 20(1), 2013.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev. ATL: A model transformation tool. *Sci. Comput. Program.* 72(1-2), 2008.

[LMRL13] Y. Lamo, F. Mantz, A. Rutle, J. de Lara. A Declarative and Bidirectional Model Transformation Approach Based on Graph Co-spans. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*. ACM, 2013.

[LWM+12] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, A. Rutle. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In *Computer and Information Science 2012*. Volume 429, pp. 37–52. 2012.

[RDCD10] L. Ribeiro, F. L. Dotti, S. A. da Costa, F. C. Dillenburg. Towards Theorem Proving Graph Grammars using Event-B. *ECEASST* 30, 2010.

[RRLW09] A. Rutle, A. Rossini, Y. Lamo, U. Wolter. A Diagrammatic formalisation of MOF-based modelling languages. *Objects, Components, Models and Patterns*, 2009.

[RRLW12] A. Rutle, A. Rossini, Y. Lamo, U. Wolter. A formal approach to the specification and transformation of constraints in MDE. *JLAP* 81(4), 2012.

[TV11] J. Troya, A. Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology* 10, 2011.

[YXF+13] L. Yngve, W. Xiaoliang, M. Florian, B. Øyvind, S. Anders, R. Adrian. DPF Workbench: a multi-level language workbench for MDE. In *Proceedings of the Estonian Academy of Sciences*. Pp. 3–15. 2013.