

# On the Use of Alloy to Analyze Graph Transformation Systems

Luciano Baresi and Paola Spoletini

Politecnico di Milano  
Dipartimento di Elettronica e Informazione  
piazza Leonardo da Vinci 32, 20133 Milano, Italy  
baresi|spoleti@elet.polimi.it

**Abstract.** This paper proposes a methodology to analyze graph transformation systems by means of Alloy and its supporting tools. Alloy is a simple structural modeling language, based on first-order logic, that allows the user to produce models of software systems by abstracting their key characteristics. The tools can generate instances of invariants, and check properties of models, on user-constrained representations of the world under analysis. The paper describes how to render a graph transformation system —specified using AGG— as an Alloy model and how to exploit its tools to prove significant properties of the system. Specifically, it allows the user to decide whether a given configuration (graph) can be obtained through a finite and bounded sequence of steps (invocation of rules), whether a given sequence of rules can be applied on an initial graph, and, given an initial graph and an integer  $n$ , which are the configurations that can be obtained by applying a sequence of  $n$  (particular) rules.

## 1 Introduction

Graphs provide the underlying structure for many artifacts produced during the development of software systems. No matter of the actual process we follow, we always end up with diagrams and models that can easily be conceived as suitably annotated graphs. For example, graphs provide a sufficiently general infrastructure to model the topology of object-oriented and component-based systems, as well as the architecture of distributed applications. This means that graph transformations are often needed —either explicitly or implicitly— to specify how these models are built and interpreted, and how they can evolve over time.

Graph transformation [5] originated in reaction to shortcomings in the expressiveness of classical approaches to rewriting, like Chomsky grammars and term rewriting, to deal with non-linear structures. The theory behind it has been evolving for some thirty years, but only recently there have been attempts to analyze modeled transformation systems. After many different modeling notations and theoretical approaches for specifying graph transformation systems, part of the research community is addressing the problem of analyzing such sets

of rules. Besides approaches, like *critical pair analysis* [1], that provide a means to discover conflicts among rules, tools like *CheckVML* [20] and *GROOVE* [11] start from conventional analysis techniques (model checking, in these cases) and exploit them to prove reachability properties on the modeled sets of rules. Typically, these proposals allow the designer to understand if a given target graph is reachable from an initial graph, through a finite number of transformation rules.

The importance of these approaches is twofold. Besides providing interesting insights from a purely theoretical point of view, they are also valuable for analyzing and discovering properties in the host domain. For example, if we used a graph transformation system to model the dynamic behavior of a software architectural style [6], or we used it to model the operational semantics of a visual notation [17], the capability of analyzing the graph transformation system allows us to discover properties on designed software architectures or on the evolution of produced models.

When the analysis approach exploits model checking, it must face and cope with the typical problems of this analysis technique: state explosion and thus limited capability of rendering the peculiarities of the modeled domain. In contrast, Alloy [16] provides a viable compromise between the richness of models and their analyzability. Alloy is a simple structural modeling language, based on first-order logic, that allows the user to produce models of software systems by abstracting their key characteristics. The SAT-based analyzer can generate instances of invariants, and check properties of models, on user-constrained representations of the world under analysis. Alloy does not address infinite words and it copes with state explosion by asking the user to specify the maximum cardinality of the worlds under analysis. The interesting features of Alloy led us to investigate the possibility of encoding a graph transformation system and use its tools to analyze reachability properties and to study the applicability of sequences of transformation rules.

This paper presents the first results of this encoding. It highlights the translation process and exemplifies it on a simple case study, taken from [1]. The paper also describes the properties on the graph transformation system that can be checked with Alloy.

The rest of the paper is organized as follows. Section 2 briefly introduces Alloy. Section 3 explains the translation of graph transformation systems into Alloy and Section 4 discusses the properties we can check on these models and how we can verify them. Section 5 surveys similar proposals and Section 6 discusses the positive and negative aspects of the approach and concludes the paper.

## 2 Alloy

Alloy is a formal notation based on relational logic, that is, a logic with clear semantics based on relations. In this section, we only introduce the key characteristics of the notation through an example; interested readers can refer to [15] for an in-depth presentation. The example models finite state automata and specifies some basic properties.

```
module Automa
```

is the module declaration.

```
sig Event{}  
sig State{}
```

are empty signatures used to introduce the concepts of `Event` and `State`, respectively.

```
sig Transition{  
  startingState: State,  
  arrivalState: State,  
  trigger: Event  
}
```

defines a `Transition` as three relations: `startingState` and `arrivalState` identify the source and target `States`, while `trigger` defines the `Event` that triggers the transition. Relations are similar to the fields of an object in the classical object-oriented paradigm.

```
sig FiniteStateAutomaton{  
  states: set State,  
  transitions: set Transition,  
  initialState: states,  
  finalStates: some states,  
  dangerousStates: set(states-initialState-finalStates)  
}
```

`states` and `transitions` are the sets of `States` and `Transitions` that belong to the automaton. `initialState` is one of the `states`, while `finalStates` are a non-empty subset (`some`) of the `states`. The `dangerousStates` are those `states` that are neither initial nor final.

After the signatures, we have the facts that constrain the instantiation of the signatures previously defined and of their relations. A fact is an explicit constraint on the model. It is possible to express constraints on the relations of a signature directly after the signature body without using the keyword `fact`.

```
fact Determinism{  
  all a:FiniteStateAutomaton| no disj t1, t2: a.transitions{  
    t1.startingState=t2.startingState  
    t1.trigger=t2.trigger}}}
```

imposes that, for any `FiniteStateAutomaton a`, there does not exist a pair of disjoint transitions in `a` such that they have the same `startingState` and the same `trigger`.

```
fact CorrectTransition{  
  all a:FiniteStateAutomaton |  
  all t: a.transitions |  
  t.startingState in a.states && t.arrivalState in a.states}
```

imposes that for any `FiniteStateAutomaton a`, the `startingState` and `arrivalState` of its `transitions` must be contained in its `states`.

Alloy comes with dedicated analysis tools: a consistency checker and a counterexample extraction tool. Both the analyses are fully automated and based on SAT solvers<sup>1</sup>. The Alloy model is translated into a boolean formula, which is then passed to the SAT solver that tries to find an assignment for all the variables in the formula to satisfy it. If the assignment exists, it is translated back into Alloy.

The consistency checker evaluates `predicates` on defined models. Predicates are like facts, but they do not affect the structure of the world under analysis; they impose “temporary” constraints whose validity is limited to the predicate they belong to.

```
pred example(){}
```

If the `predicate` is empty, as the above `example`, the Alloy analyzer checks the consistency of the model itself (with no further constraints).

A predicate is verified through a `run`, which tries to find an assignment that satisfies the model, along with the constraints of the predicate under analysis.

```
run example for 1 FiniteStateAutomaton, 2 Transition, 3 State, 2 Event
```

When we `run` a model, we must specify the maximum cardinality of the sets of the world under analysis. We can supply a unique cardinality for all the sets, but we can also associate special values with particular sets. In this case, `example` considers exactly 1 `FiniteStateAutomaton`, 2 `Transitions`, 3 `States`, and 2 `Events`.

The identification of counterexamples is performed through `asserts`, which claim that something must be true due to the behavior of the model.

```
assert isolation{
all a:FiniteStateAutomaton| #a.states>1 =>
  all s:a.states| some t:Transition|
    s = t.startingState || s = t.arrivalState}
```

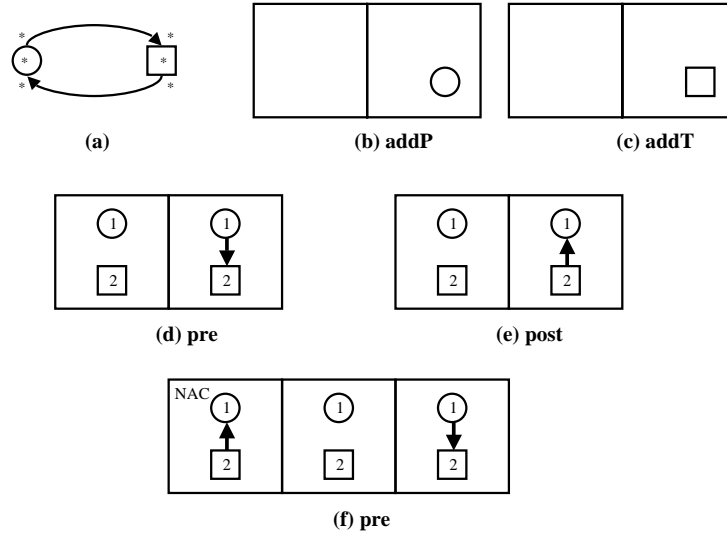
```
check isolation for 5
```

states that there is no state in any automaton with at least two states that is not the initial or the final state of a transition of the automaton. Assertions are checked by searching for counterexamples, using `check` commands and again we need to set the upper-bound for the cardinalities of the sets that define the world under analysis. In this case, we use a single value, and we only consider sets of five elements. The assertion generates a counterexample since there is no constraint on the automaton that limits the number of isolated states.

Alloy is targeted to describing and analyzing structural properties of systems, but Jackson et al. [16] introduce also a mechanism to represent traces. A more

---

<sup>1</sup> Alloy works with different SAT solvers with different characteristics.



**Fig. 1.** Graph transformation system to create place-transition nets

general approach to deal with execution traces is presented in [10]: DynAlloy is a dynamic version of Alloy for modeling the execution of operations and reasoning about execution traces. Alloy specifications can also be verified by using theorem proving techniques: Arkoudas et al. [2] present Prioni, a tool that uses the semi-automatic theorem prover Athena to prove properties regarding Alloy specifications.

### 3 Encoding

This paper uses the notation proposed by AGG [7] to model graph transformation systems. For example, Fig. 1 shows the simple rules needed to build correct place-transition nets, a particular instance of finite state automata. More precisely a place-transition net is a tuple  $(P, T, \pi, \tau)$ , where  $P$  is a set of places,  $T$  is a set of transitions,  $\pi : P \rightarrow T$  and  $\tau : T \rightarrow P$  are functions that associate a transition to a place and a place to a transition, respectively.

Fig. 1(a) shows the type graph, i.e., the meta-model of the system, and identifies two components: places, the circle, and transitions, the square; places can be connected to a finite number of transitions and vice versa. All the other parts of the figure represent the rules: Fig. 1(b) represents the rule to add a place to the net, Fig. 1(c) the rule to add a transition, and Fig. 1(d) and Fig. 1(e) the rules to connect a place to a transition and a transition to a place, respectively. Since the presented example does not contain negative application conditions, Fig. 1(f) shows the modification of the rule of Fig. 1(d) to add the constraint

that to add a relation from a place to a transition, we require that no inverse relations exist.

Our use of Alloy aims at representing the evolution of a system, described as a graph, through the application of the rules that compose the transformation system. To this end, we introduce the following signature —borrowed from [16]— to represent graph paths:

```
sig Path{
  elem: set Graph,
  first, last: elem,
  next:(elem-last) one -> one (elem-first)
} {
  first!=last
}
```

The path itself represents the evolution of the system. A path is a set of graphs that goes from an initial graph, `first`, to the graph, denoted as `last`, reached by applying  $|elem| - 1$  transformation rules.

The relation `next` assigns exactly one element to each graph different from the final graph. The `fact2 (first!=last)` ensures that the path is composed of at least a transformation, imposing that the initial and final graphs must be different. This restriction guarantees that all the elements in the path are connected; in fact if initial and final graphs were the same, the path would be disconnected.

This signature is independent of the context and it is added to the Alloy model for any graph transformation system. The composition of the path and the rules used to build the system, instead, are defined by means of a fixed translation process that depends on the particular graph transformation system.

Specifically, we introduce a signature `Graph` along with a signature for all the elements that compose the meta-model. The newly introduced `Graph` contains a relation to each of these signatures. The cardinality of this relation depends on the cardinalities that the elements have in the *type graph*. In our example of Fig. 1(a), the meta-model contains two types of elements, nodes and transitions, both with multiple cardinality. Hence, the `Graph` is defined in Alloy as follows:

```
sig Graph{
  places: set Place,
  transitions: set Transition
}
```

The signatures representing the elements are composed of two parts: attributes and connections. The first contains a unary relation for each attribute of the element. Appropriate signatures are introduced to render the types ascribed to the different elements. Particular attention must be paid to integers and booleans: the former are a *built-in* concept in Alloy, while the latter can be

---

<sup>2</sup> Notice that in this case, we use the compact form to represent facts within the signatures they belong to.

represented using a **lone** (zero or one) relation to a mirror signature with the empty relation that corresponds to *false*. The connection part contains a set of relations, one for each connection in the *type graph*. The relations are added only to the source elements of the associations in the meta-model. The cardinality of these relations is given based on the cardinality of the target association end. The cardinality of the source association end is used to constrain the system. For instance, if the cardinality of the target association end between elements  $e_1$  and  $e_2$  is 1, this means that we cannot have two instances of  $e_1$  related to the same instance of  $e_2$ . These constraints are added as **facts**.

In our example, the elements have no attributes, but they have outgoing arcs, and thus each of them has a relation. The arcs (associations) are marked with an **\*** on both sides, which become **sets** in the Alloy model, with no constraints added to the signatures:

```
sig Place{
    enable: set Transition
}
sig Transition{
    fireTo: set Place
}
```

The rules of the graph translation system, which represent the rules to build the edges in signature **Path**, are modeled in Alloy as **predicates**. For each graph transformation rule, we introduce a predicate with the following parameters: a) Two graphs,  $g_1$  and  $g_2$ , represents the LHS and the RHS of the production, b) two parameters for each element that is modified correspond to the old and new values, c) one parameter for each element that is deleted from the LHS or added to RHS. The RHS graph  $g_2$  is obtained as follows:

$$g_2.r_i = g_1.r_i - \Sigma_j eLHS_j^{mod} - \Sigma_k eLHS_k^{canc} + \Sigma_j eRHS_j^{mod} + \Sigma_h eRHS_h^{add}$$

where  $eLHS_j^{mod}$ ,  $eLHS_k^{canc}$ ,  $eRHS_j^{mod}$ , and  $eRHS_h^{add}$  are all the elements of the predicates of same type of the right-hand side of relation  $r_i$ .  $eLHS_j^{mod}$  represents the element that has to be modified as it appears in the LHS,  $eRHS_j^{mod}$  the element modified as it appears in the RHS,  $eLHS_k^{canc}$  the element in the LHS that is cancelled in the RHS, and  $eRHS_h^{add}$  the element added in the RHS.

We also add a constraint to ensure that all the elements that appear in the RHS and not in LHS, are not part of the graph that represents the left hand side. The predicates also contain the characteristics that the elements have in terms of attributes and relations, and the connections between  $eLHS_j^{mod}$  and  $eRHS_j^{mod}$ .

The representation in Alloy of the AGG rules of Fig. 1 is the following.

```
pred addP(g1,g2:Graph, p:Place){
    g2.transitions=g1.transitions &&
    g2.places=g1.places+p &&
    p not in g1.places &&
    #p.enable=0
}
```

Rule `AddP` adds a place to the RHS, hence it is translated into a predicate that has the initial and the final graphs and the added place as parameters. Since this place has no relations with any transition, it is imposed that the number of elements (`#`) in relation `enable` for `p` is zero. Then, since the only other parameter, besides the graphs, is place `p`, the transitions of the two graphs are the same, while the places of the final graph are the union of those of the initial graph and `p`, which must not be present in `g1`. Rule `addT` is the same as `addP`, but it adds a transition.

```

pred addT(g1,g2:Graph, t:Transition){
    g2.transitions=g1.transitions+t &&
    g2.places=g1.places &&
    t not in g1.transitions &&
    #t.fireTo=0
}

```

Rules `pre` and `post` are more complex.

```

pred pre(g1,g2:Graph, p1,p2:Place,t:Transition){
    g2.transitions=g1.transitions &&
    g2.places=g1.places-p1+p2 &&
    p2 not in g1.places &&
    p2.enable=p1.enable+t
}

pred post(g1,g2:Graph, p:Place, t1,t2:Transition){
    g2.transitions=g1.transitions-t1+t2 &&
    g2.places=g1.places &&
    t2 not in g1.transitions &&
    t2.fireTo=t1.fireTo+p
}

```

Since the two rules are symmetric, we only consider the first one. This rule involves a place and a transition, the place is modified, while the transition is not. Hence, the parameters are the two graphs, two places, and a transition. No particular constraints are imposed on `t`, while the set of places in the final graph must differ from the one in the initial graph for a place. In fact, place `p1` is replaced by `p2`, which is related to all the transitions in relation `p1.enable`, with the addition of `t`. Moreover `p2` must not appear in the original graph.

When a graph transformation rule has a negative application condition, this constraint must be represented in the Alloy predicate. The NAC is a precondition on the elements involved in the transformation. Generally, these elements are already parameters of the predicate; if not, they are added. Then, for each element in the negative application condition, we add a formula to constrain involved elements not to assume the forbidden configuration. As example, we can consider rule `pre` of Fig. 1(f), where the NAC imposes that we cannot add a relation from a place to a transition if the opposite relation already exists. Notice that this does not mean that place-transition nets are acyclic or they have not

cycle of length 2 (place – transition – place), but that, if such a cycle exists, it was built by creating the association between a place and a transition before its opposite. The Alloy predicate is modified as follows:

```

pred pre(g1,g2:Graph, p1,p2:Place, t:Transition){
  g2.transitions=g1.transitions &&
  g2.places=g1.places-p1+p2 &&
  p2 not in g1.places &&
  p2.enable=p1.enable+t &&
  p1 not in t.fireTo
}

```

where the formula `p1 not in t.fireTo` is the representation of the NAC.

After defining the predicates for all the rules of the graph transformation system, we want to impose that these rules are the only way to move from a configuration of the graph to another, that is, they are the only way to relate two elements in the `Path` with respect to relation `next`. To model this constraint in Alloy, we add an additional constraint to the facts regarding signature `Path` to impose that for relation `next`, one of the predicates holds, that is, the edge is obtained by applying one of the rules of the system. In our example the constraint is:

```

all e,e':elem| (e in e'.next =>
  ((one p:e.places| addP(e',e,p))||
  (one t:e.transitions| addT(e',e,t))||
  (one p1:e'.places| one p2:e.places|
    one t:e.transitions| pre(e',e,p1,p2,t))||
  (one t1:e'.transitions| one t2:e.transitions|
    one p:e.places| post(e',e,p,t1,t2))))

```

and states that for each pair of adjacent graph configurations, there exists exactly one place that is added to the first configuration to obtain the second through production `addP`, or exactly a transition is added to the first configuration to create the second through `addT`, or the second configuration is obtained by adding a relation from a place to a transition, or from a transition to a place, to the first configuration.

Even if presented through an example, the encoding is general and algorithmic, and thus can be applied on other and more complex transformation systems.

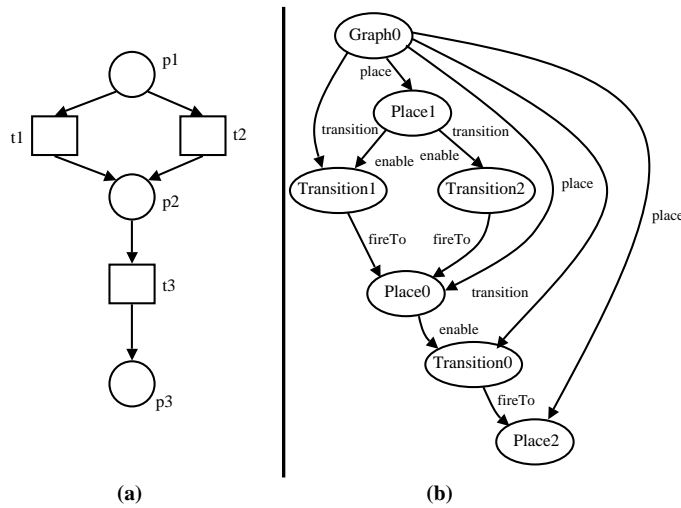
## 4 Verification

The tools allow us to check the reachability of given configurations of the host graph through a finite sequence of steps (invocations of rules), to verify whether given sequences of rules can be applied on an initial graph, and to show all the configurations that can be obtained by applying a sequence of `n` (particular) rules on a given initial graph. Moreover, particular systems might motivate us

to verify the validity of particular user-defined properties (constraints) on valid configurations.

Alloy allows us to try whether a property can be satisfied or to show that it does not hold. The property is translated into a predicate, in the former case, or its negation is translated into an assertion, in the latter case.

The reachability of configurations allows us to check whether a given configuration can be reached through a finite set of transformation rules. This property can be applied to any user-defined configuration; the default one is the empty graph.



**Fig. 2.** Example initial graph configuration and a solution produced by Alloy

If we consider the graph of Fig. 2(a) and we want to verify that it is reachable from the the empty graph, we can build the following predicate:

```
pred reachConfig(){
  some p:Path| initialGraph(p.first) && DefConfig(p.last)
}
```

where `initialGraph(p.first)` and `defConfig(p.last)` are two predicates representing the initial and desired final configurations, respectively. In our example, they are defined as follows (but they might be much more complex):

```
pred initialGraph(g:Graph){
  #g.places=0 && #g.transitions=0
}
```

```
pred defConfig(g:Graph){
```

```

#g.places=3 && #g.transitions=3 &&
one p1,p2,p3:g.places| some disj t1,t2,t3:g.transitions|
p1.enable=t1+t2 && t1.fireTo=p2 && t2.fireTo=p2 &&
p2.enable=t3 && t3.fireTo=p3 && #p3.enable=0
}

```

Fig. 2(b) shows an instance of the model that satisfies `pred defConfig()`. The model is a set of nodes that represent the different instances of the signature; edges represent the relations among them. For example, `Graph0` embeds place `Place1`, which is related, through `enable`, to `Transition1` and `Transition2`.

If we want to check that a given configuration is not reachable, we must check the following assertion:

```

assert unreachConfig(){
  no p:Path| initialGraph(p.first) && defConfig(p.last)
}

```

The validation of a sequence of rules allows us to verify that, starting from an initial graph, a given sequence of rules is applicable on it. To verify the property, we have to introduce a predicate with the following structure:

```

pred validRules(){
  some p:Path| some disj t1,...,tk:Type1| ... | some disj f1,...,fh:Typej|
  givenConfig(p.first) &&
  R1(p.first, p.next[p.first], parR1) &&
  ...
  Rn(p.next[p.next...[p.first]],p.next[p.next[p.next...[p.first]]],parRn)
}

```

where `givenConfig()` is a predicate that represents the starting graph, `some disj t1,...,tk:Type1| ... | some disj f1,...,fh:Typej` identifies the variables needed, besides the graph, in the predicates that represent the rules to be applied and `R1, ..., Rn` are the rules of the sequence we want to verify.

As an example, the verification of the sequence `addT, addP, pre` from the initial graph is performed through the following predicate:

```

pred validRules(){
  some p:Path| some disj t1,t2:Transition| some disj p1,p2,p3:Place
  givenConfig(p.first) &&
  addT(p.first, p.next[p.first], t1) &&
  addP(p.next[p.first], p.next[p.next[p.first]], p1) &&
  pre(p.next[p.next[p.first]],p.next[p.next[p.next[p.first]]], p2, p3, t2)
}

```

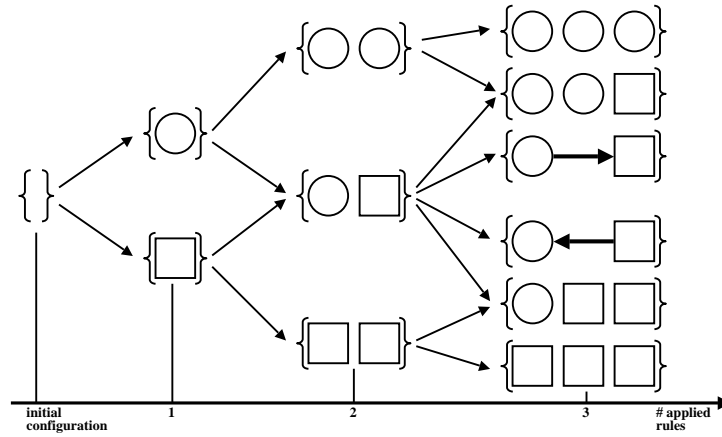
Finally, to analyze reached configurations, that is, to show all the possible configurations obtained by means of a generic allowed path of a specified length from an initial state, we need to use one of the SAT solvers embedded in Alloy that allows us to find multiple solutions (e.g., MCHAFF). Moreover, since this SAT solver shows all the possible configurations that satisfy the property, it is necessary to constrain the assertion we use to obtain “only” the paths we want. The general structure of this property is the following:

```

pred configLength_n(){
  some p:Path| p.elem=n+1 && givenConfig(p.first) &&
  one Path && no(A1-p.elem.a1) && ... &&
  no(Am-p.elem.am)
}

```

where  $n$  is the desired length for the path,  $a_1, \dots, a_m$  represent the relations in Graph that are used for the path, and  $A_1, \dots, A_m$  are the corresponding signatures.



**Fig. 3.** All possible paths

Hence, if we wanted to extract all the possible paths of length 3 from the initial configuration in our example, the predicate is the following:

```

pred ConfigLength_3(){
  some p:Path| #p.elem=4 &&
  one Path && initialGraph(p.first) &&
  no(Transition-p.elem.transitions) &&
  no(Place-p.elem.places)
}

```

The run of this property allows us to find all the possible paths of length 3 in the given graph transformation system starting from the empty configuration. The result is presented in Fig. 3.

## 5 Related work

The analysis of graph transformation systems can be performed in different ways. Heckel in [13] gives the theoretical foundations for the verification of graph

transformation systems through model checking: graphs are interpreted as states and rule applications as transitions. This idea is exploited both by GROOVE [18] and by checkVML [20].

GROOVE [18] is based on the idea of using the core concepts of graphs and graph transformations all the way through during model checking. States are represented as graphs, while transitions are represented as applications of graph transformation rules. This way, properties are specified in a graph-based logic to apply graph-specific model checking algorithms. Hence, in this approach, only some ideas of traditional model checkers can be applied immediately, since the most basic concept, namely the underlying model, has been extended drastically.

On the other side, CheckVML [20] exploits off-the-shelf model checker tools, like SPIN [14], to verify graph transformation systems. More thoroughly, CheckVML takes as input a graph transformation system parameterized with a type graph and an initial graph (represented by means of an abstract transition system) and gives an equivalent model in Promela, the input language of SPIN. Property graphs are also translated into their temporal logic equivalents.

In [19], Rensink et al. propose a comparison between these two tools, and conclude that CheckVML always performs better when dynamic allocation and symmetries are limited, while for dynamic problems GROOVE is preferable.

VIATRA [8] is another graph transformation tool with interesting capabilities for controlling the transformation and composition of complex transformations. The graph transformations are driven by abstract state machines, as specification formalism; extended hierarchical automata represent the model. In the end, VIATRA checks statecharts with SPIN.

Baldan and König [4] describe a different theoretical framework. It aims at analyzing a special class of hypergraph rewriting systems by a static analysis technique based on approximative foldings and unfoldings of a special class of Petri nets. Baldan et al. [3] extend this work by providing a precise (McMillan-style) unfolding strategy. Dotti et al. [9] use object-based graph grammars for modeling object-oriented systems and define a translation into the input language of SPIN to use model checking. The authors allow a restricted structure for graph transformation rules tailored to the message call mechanism of object-oriented systems. Even if the chosen representation in SPIN only supports a restricted problem, the structure of the generated code, in general, results in better run-time performance.

## 6 Conclusions and future work

The paper presents a proposal for exploiting the formal language Alloy to analyze graph transformation systems. We present the first ideas behind the encoding process and we demonstrate them on a simple case study. Besides the example in the paper, we applied the proposed methodology on the Concurrent Append example presented in [19], and on the Shopping example shown in [12], and we obtained encouraging results.

The main drawback of this approach, with respect to GROOVE and Check-VML, is the need for tailoring the search space. In fact, these methods utilize model checking techniques, and the whole space of the modeled system is searched automatically —if the model checker does not go out of memory. In our case, with Alloy, we only analyze the system for a finite scope, whose size is user-defined. Even if this feature may look like a limitation, it is true that this way users have much more freedom to tailor the details embedded in the Alloy models they want to analyze with respect to the size of the spaces they want to deal with. It is also true that the SAT-based analysis techniques produce interesting results faster and by using less memory.

We do not want to say that the proposed approach is better than those that employ model checking; we only want to let the reader know of other options for analyzing designed graph transformation systems, fostering the capability of specifying reach models, which are analyzable because of the finite size of the search space, and the SAT-based techniques.

We think that our method is useful to find instances and counterexamples for models and graph transformation systems. In general, we are not interested in infinite paths, but it is interesting to analyze that our properties are verified by applying a certain number of transformation rules and with a “big enough” model. Heuristics and practical cases suggest us that it is always possible to identify a scope that is big enough to let us explore the interesting parts of systems.

In the future, we plan to improve the approach and implement a component to automatically translate a graph transformation system described using AGG into an Alloy model. This would be the natural conclusion of the first phase of the research, which is investigating the best encoding heuristics. It is the only way to conduct more significant experiments and refine the properties we are interested in. We also have plans to integrate our approach with Prioni, and thus exploit the analysis capabilities of a theorem prover.

## References

1. AGG. <http://tfs.cs.tu-berlin.de/agg/>.
2. K. Arkoudas, S. Khurshid, D. Marinov, and M. Rinard. Integrating model checking and theorem proving for relational reasoning. In *Proceedings of the Seventh International Seminar on Relational Methods in Computer Science (ReMiCS 2003)*, volume 3015 of *Lecture Notes in Computer Science (LNCS)*, pages 21–33, 2003.
3. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: An unfolding-based approach. In *Proceedings of CONCUR 2004*, pages 83–98, 2004.
4. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In *Proceedings of ICGT*, pages 14–29, 2002.
5. L. Baresi and R. Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 402–429. Springer-Verlag, 2002.
6. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proceedings of European Software*

- Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 68–77. ACM Press, 2003.
7. M. Beyer. *AGG1.0 - Tutorial*. Technical University of Berlin, Department of Computer Science, 1992.
  8. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. Viatra - visual automated transformations for formal verification and validation of uml models. In *Proceedings of ASE*, pages 267–270, 2002.
  9. F.L. Dotti, L. Foss, L. Ribeiro, and O. Marchi dos Santos. Verification of distributed object-based systems. In *Proceedings of FMOODS*, pages 261–275, 2003.
  10. M. Frias, C. Lopez Pombo, G. Baum, N. Aguirre, and T. Maibaum. Reasoning about static and dynamic properties in alloy: A purely relational approach. *ACM Trans. Softw. Eng. Methodol.*, 14(4):478–526, 2005.
  11. GROOVE. <http://groove.sourceforge.net/groove-index.html>.
  12. J.H. Hausmann, R. Heckel, and G. Taentzer. Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In *Proceedings of ICSE*, pages 105–115, 2002.
  13. R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *Proceedings of FASE*, pages 138–153, 1998.
  14. G.J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
  15. D. Jackson. *Software Abstractions : Logic, Language, and Analysis*. The MIT Press, 2006.
  16. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 62–73. ACM Press, 2001.
  17. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *Proceedings of UML 2001*, volume 2185 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
  18. A. Rensink. The GROOVE simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.
  19. A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.
  20. Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, San Francisco, CA, USA, October 20-24 2003. Springer.