The Graph Transformation Benchmark for the International Knowledge Engineering Competition for Planning & Scheduling

Stefan Edelkamp
Computer Science Department
University of Dortmund
stefan.edelkamp@cs.uni-dortmund.de

Arend Rensink
University of Twente
rensink@cs.utwente.nl

March 6, 2007

1 Introduction

This document provides the material necessary to deal with the graph transformation benchmarks, as featured in the International Competition on Knowledge Engineering for Planning & Scheduling (ICKEPS-07). We propose GROOVE as a simulator to study different test models.

Graph transition systems are suitable representations for software and hardware systems and extend traditional transition systems by relating states with graphs and transitions with partial graph morphisms. Intuitively, a (partial) graph morphism associated to a transition represents the relation between the graphs associated to the source and the target state of a transition. More specifically, it models the merging, insertion, addition and renaming of graph items (nodes or edges).

The GROOVE project takes the point of view that graphs are a very good basis for both design-time and run-time models of (software) systems. Graphs

have the advantage of a visual representation (even though this tends to remain practical only for small-scale examples) and, more importantly, a rich formal foundation; in addition, they are flexible enough to deal with all kinds of models without a priori constraining them. Finally, in the long-standing theory of graph transformation we find a mathematical tool to formalize many, most, or even all of the types of transformation discussed above.

2 Graph Transition Systems

Competitors that their planners and tools do not have operate on the very generic domain of graph transition systems but work on the individual graph transformation domains instead. The declaration of the according graph transformation systems will be made available to the competitors.

In order to get a flavor, what graph transformation is about, following set of definitions briefly introduces the theory for the *single-pushout* approach based on partial graph morphisms with a left and right rule application pair[1].

Definition 1 *A (*multigraph*) G is a tuple $\langle V_G, E_G, src_G, tgt_G \rangle$ where V_G is a set of nodes, E_G is a set of edges, $src_G, tgt_G : E_G \to V_G$ are a source and target functions.*

Graphs usually have a distinguished start state which we denote with s_0^G, or just s_0 if G is clear from the context.

Definition 2 *A* path *in a graph G is an alternating sequence of nodes and edges represented as $u_0 \xrightarrow{e_0} u_1 \ldots$ such that for each $i \geq 0$ we have $u_i \in V_G$, $e_i \in E_G$, $src_G(e_i) = u_i$ and $tgt_G(e_i) = u_{i+1}$, or, shortly $u_i \xrightarrow{e_i} u_{i+1}$.*

An initial path is a path starting at s_0^G. Finite paths are required to end at states. The length of a finite path p is denoted by $|p|$. The concatenation of two paths p, q is denoted by pq, where we require p to be finite and end at the initial state of q.

[1]For the *double-pushout* approach each rule consists of a triple of left-hand side, invariant and a right-hand side. A rule specifies that an occurrence of the left-hand side L in a larger graph G can be rewritten into the right hand side R preserving the interface K. For more information see [1].

Definition 3 *A* graph morphism $\psi : G_1 \to G_2$ *is a pair of mappings* $\psi_V : V_{G_1} \to V_{G_2}$, $\psi_E : E_{G_1} \to E_{G_2}$ *such that we have* $\psi_V \circ src_{G_1} = src_{G_2} \circ \psi_E$, $\psi_V \circ tgt_{G_1} = tgt_{G_2} \circ \psi_E$.
A graph morphism $\psi : G_1 \to G_2$ *is called* injective *if so are* ψ_V *and* ψ_E; identity *if both* ψ_V *and* ψ_E *are identities, and isomorphism if both* ψ_E *and* ψ_V *are bijective. A graph* G' *is a* subgraph *of graph* G, *if* $V_{G'} \subseteq V_G$ *and* $E_{G'} \subseteq E_G$, *and the inclusions form a graph morphism.*

A partial graph morphism $\psi : G_1 \to G_2$ *is a pair* $\langle G'_1, \psi_m \rangle$, *where* G'_1 *is a subgraph of* G_1, *and* $\psi_m : G'_1 \to G_2$ *is a graph morphism.*

The composition of (partial) graph morphisms results in (partial) graph morphisms. Now, we define a notion of transition system.

Definition 4 *A* transition system *is a graph* $M = \langle S_M, T_M, in_M, out_M \rangle$ *whose nodes and edges are respectively called* states *and* transitions, *with* in_M, out_M *representing the source and target of an edge respectively.*

Finally, we are ready to define graph transition systems, which are transition systems together with morphisms mapping states into graphs and transitions into partial graph morphisms.

Definition 5 *A* graph transition system *(GTS) is a pair* $\langle M, g \rangle$, *where* M *is a transition system and* $g : M \to \mathcal{U}(\boldsymbol{G}_p)$ *is a graph morphism from* M *to the graph underlying* $\boldsymbol{G}_p$, *the category of graphs with partial graph morphisms. Therefore* $g = \langle g^S, g^T \rangle$, *and the component on states* g^S *maps each state* $s \in S_M$ *to a graph* $g^S(s)$, *while the component on transitions* g^T *maps each transitions* $t \in T_M$ *to a partial graph morphism* $g^T(t) : g^S(in_M(t)) \Rightarrow g^S(out_M(t))$.

3 The GROOVE Simulator

The GROOVE project [9] aims at the usage of model checking techniques for verifying object-oriented systems, where the states of the system are modeled as graphs, instead of bit vectors as in most explicit state representing approaches. This approach creates new opportunities to specify and verify systems in which the states mainly depend on a set of reference values instead of values of primitive types (with a finite domain) only. Due to frequent (de)allocation of reference values, the states of such systems are highly

dynamic, due to their variable size. Graphs provide a natural way of representing the states of such systems and specifying interesting properties. The tool follows the single-pushout approach as introduced above.

The GROOVE simulator [12] does a small part of the job of a model checker: it attempts to generate the full state space of a given graph grammar. This entails recursively computing and applying all enabled graph production rules at each state. Each newly generated state is compared to all known states up to isomorphism; matching states are merged, in the way proposed in [13]. No provisions are currently made for detecting or modeling infinite state spaces. Alternatively, one may choose to simulate productions manually. The tool is designed for extensibility. The internal and visual representation of graphs are completely separated, and interfaces are heavily used, for instance to abstract from graph and morphism implementations.

The simulator is implemented in Java. The tool can handle arbitrary production rules, but can obviously only generate a finite part of the corresponding graph transition system. The simulator uses non-attributed, edge-labeled graphs without parallel edges. Node labels are actually labels of self-edges. The most performance-critical parts of the simulator are: *finding rule matchings*, and *checking graph isomorphism*. The first problem is, in general, NP-complete; the second is in NP (its precise complexity is unknown). Fortunately, the graphs we

The modularity of GROOVE also extends to the serialization and storage of graphs and graph grammars. Currently the tool uses GXL [16], but in an ad hoc fashion: production rules are first encoded as graphs and then saved individually; thus, a grammar is stored as a set of files. A sample GXL specification is provided in the Appendix.

Some performance figures on GROOVE were reported in [12]. Two intrinsically complex parts of the state space generation are: finding occurrences of left hand sides, and determining isomorphism of states. GROOVE is made available for the public at `http://sourceforge.net/projects/groove`.

4 Modeling Planning Problems in GROOVE

Graph transformation systems enable users to encode domain models including states and transition rules in form of graphs.

This section gives insights and examples of applying graph transformation techniques for specifying and simulating action planning domains in a

graphical interface. We further illustrate the impact of planners in solving some graph transition problems faster than with current technology. For the sake of simplicity, we restrict to propositional planning even if GROOVE has been recently extended to deal with numeric attributes.

Action planning refers to a world description in logic. A number of atomic propositions describe what can be true or false in each state of the world. By applying operations in a world, we arrive at another world where different atoms might be true or false. Usually, only some few atoms are affected by an action, and most of them remain the same. A concise representation the STRIPS formalism [4], an acronym for an early planning system developed at Stanford University. STRIPS planning assumes a closed world. Everything that is not stated as being true is assumed to be false. Therefore, the denotation of the initial state *Init* is shorthand for the value assignment to *true* of the propositions that are included in *Init*, and to *false* of the ones that are not.

4.1 Blocksworld

In Blocksworld a robot tries to reach a target state by actions that stack and unstack blocks, or put them on the table. In Fig. 1 we show a GROOVE model the four operations *stack*, *unstack*, *pickup*, and *putdown*. Nodes represent domain objects (in this case there is only one) and edges represent attribute or predicates that combine objects. Edges attached with `new` correspond to add effects, while edges labeled by `del` abbreviate delete effects. Delete edges together with persistent edges (none for the Blocksworld instance) are the precondition of the rule.

Figure 2 illustrates the specification of the initial state in GROOVE. We find block B on top of block B and a singleton block A on the table. The robot arm is empty. Finally, Figure 3 depicts the (entire) graph transition system that is generated by continousely applying the 4 rules starting from the initial state, where rule application is established by matching the rule to the graph representing the current state.

4.2 Logistics

An example for a more complex STRIPS domain is Logistics. The task is to transport packages within cities using trucks, and between cities airplanes. Locations within a city are connected and trucks can move between any two

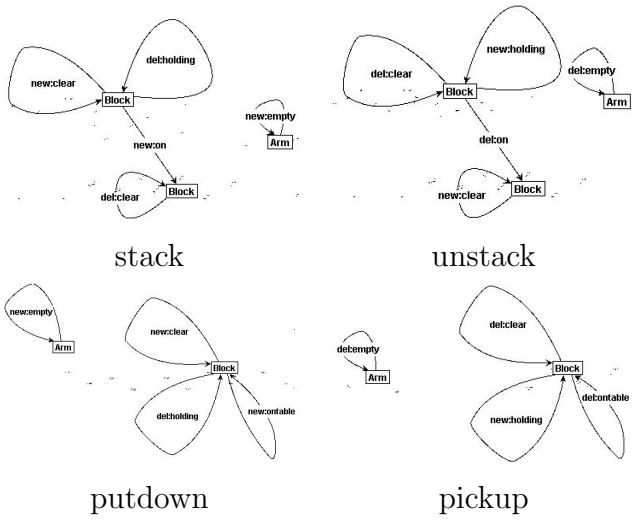

Figure 1: The graph transformation rules for Blocksworld.

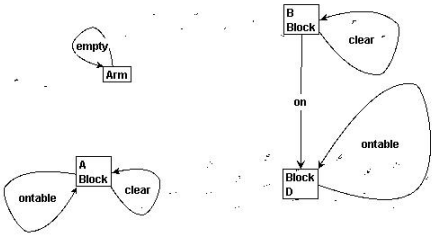

Figure 2: The initial state for Blocksworld.

such locations. In each city there is exactly one truck, each city has one airport.

Fig. 4 shows the modeling of the actions as a graph transformation rules in GROOVE. With Trucks, Airplane, Location and Packages we used four domain object types. We find context edges that do not change, like the *in* for the *drive* action.

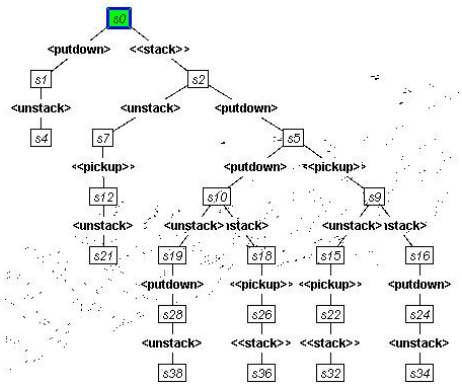

Figure 3: The state space generated from the initial state by the rule set.

5 Running a Planner on GROOVE Models

In this section we address the capability of modern planning in solving one of the problems that have been denoted as a challenge to graph transformation [14].

5.1 The Girl's Gossip Problem

We are given a number of n girls, each of which has her own secret and given a call action. On a call both participating girls divulge all the secrets they know. The question is, what is the minimal number of alls after which all girls know all secrets. The know optimum is $2n - 4$ calls.

The problem has been modeled as a graph transition system. In the following, we give an intuitive PDDL description for it utilizing conditional effects and bounded quantification as available in PDDL Level 1 /ADL. For the only action specification we have

```
(:action call
      :parameters (?g1 ?g2 - girl)
      :effect
      (and (forall (?s - secret)
               (when (has-secret ?g1 ?s) (has-secret ?g2 ?s)))
           (forall (?s - secret)
               (when (has-secret ?g2 ?s) (has-secret ?g1 ?s)))))
```

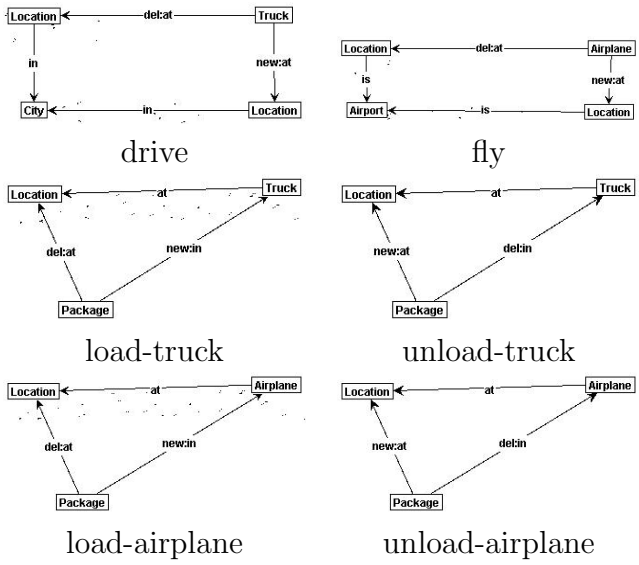

Figure 4: The graph transformation rules for Logistics.

The goal condition is that all girls know all secrets and corresponds to the requirement

```
(:goal
 (and
  (forall (?g - girl)
     (forall (?s - secret)
       (has-secret ?g ?s)))))
```

Table 1 shows the result of running a planner compared to two documented versions of GROOVE to illustrate the exponential impact of heuristic search state space enumeration[2]. In the heuristic search planner FF [6] we choose best-first instead of enforced hill-climbing as, as the former delivers plans that are non-optimal. We haven't applied A*, but an inadmissible weighted version with a heuristic weight factor of 2. In general there is no guarantee that the established solution is optimal. As we know the number of steps, we could validate that the solutions were indeed optimal.

[2]The experiments were run on a Windows Laptop with 512 MB and 3 GHz Pentium 4 processor.

	Groove Plain		Groove Quantified		FF	
# girls	# states	# sec	# states	# sec	# states	# sec
5	381	2	.	1	.	1
6	4,448	11	.	1	.	1
7	80,394	240	.	1	.	1
8	2,309,763	13,308	60,990	1,400	.	1
9	-	-	2,132,210	87,302	.	1
11	-	-	-	-	635	1
21	-	-	-	-	5170	39

Table 1: Comparison of Groove with planner on the girl's gossiping problem.

5.2 The Dining Philosophers

In Dijkstra's *dining philosophers problem* n philosophers sit around a table to have lunch. There are n plates, one for each philosopher, and n forks located to the left and to the right of each plate. Since two forks are required to eat the spaghetti on the plates, not all philosopher can eat at a time. Moreover, no communication except taking and releasing the forks takes place. The task is to devise a local strategy for each philosopher that lets all philosophers eventually eat. The simplest solution to access the left fork followed by the right one, has an obvious problem. If all philosopher wait for the second fork to be released there is no possible progress; a dead-end has occurred.

By studies in (program) model checking [3] we know that heuristics scale-up the search for the deadlock. Using an automated translation from Promela inputs (the input language of the model checker SPIN), one can derive a PDDL for the philosopher model fully automatically. This benchmark domain has been used in the 4th International Planning Competition [5]. Here, we provide a simplified PDDL model for the dining philosophers that is equivalent to the one that is provided in Groove.

```
(:action get-left
   :parameters (?p1 ?p2 - phil ?f - fork)
   :precondition
   (and (right ?p1 ?f) (left ?p2 ?f) (hungry ?p2) (not (hold ?p1 ?f)))
   :effect (and (hasLeft ?p2) (hold ?p2 ?f) (not (hungry ?p2))))
```

```
(:action get-right
   :parameters (?p1 ?p2 - phil ?f - fork)
   :precondition
   (and (right ?p1 ?f) (left ?p2 ?f) (hasLeft ?p1) (not (hold ?p2 ?f)))
   :effect (and (not (hasleft ?p1)) (hold ?p1 ?f) (eat ?p1)))

(:action go-hungry
   :parameters (?p - phil)
   :precondition (and (think ?p))
   :effect (and (not (think ?p)) (hungry ?p)))

(:action release-left
   :parameters (?p - phil ?f - fork)
   :precondition (and (eat ?p) (hold ?p ?f) (left ?p ?f))
   :effect (and (not (hold ?p ?f)) (not (eat ?p)) (hasright ?p)))

(:action release-right
   :parameters (?p - phil ?f - fork)
   :precondition (and (hold ?p ?f) (right ?p ?f) (hasright ?p))
   :effect (and  (think ?p) (not (hold ?p ?f)) (not (hasright ?p)))
```

6 Competition Usage

This section is the manual of the simulator used in the competition. The simulator is implemented in Java. The section successively describes the installation and invocation of the simulator, as well as the content of the input files and of the simulation report produced.

6.1 Installation

Download the appropriate file `GROOVE-ICKEPS.tgz`, uncompress, unpack and you are ready to start. The `GROOVE-ICKEPS` directory obtained contains the sources of GROOVE, this document `groove.pdf`, and a directory containing sample problems and plans.

6.2 Running the Simulator

The simulator is invoked as follows:

```
GROOVE-ICKEPS <plan> <prefix> <directory>
```

Where: `<rules>` is the file containing the plan description in form of a sequence of graph transformation rules, `<prefix>` is an filename prefix for the generated output files, and directory is the directory of the input files, describing the rules and initial graph of the graph transformation problem.

The simulator simulates the plan on the graph transformation problem and prints a report of the simulation to some files starting with prefix.

The ICAPS executable is implemented of a wrapper in form of a bash-script:

```
#!/bin/sh
source ../inc_paths.sh
export SIMPATH=$MODELPATH'graph'
cd $MODELPATH

java -classpath $SIMPATH/groove-edelkamp.jar:
$SIMPATH/lib/castor-0_9_5_2.jar:
$SIMPATH/lib/jgraph-5_2_1.jar:
$SIMPATH/lib/xerces-2_6_0-xercesImpl.jar:
$SIMPATH/lib/jgraphaddons-1_0_4.jar:
$SIMPATH/lib/junit-4.2.jar
groove.Generator -v 0 -x controlled:$1 -f $2 $3
```

Most problems are going to be optimization problems minimizing the number of actions or another plan objective function, but we do not allow plans ever to be partially ordered.

The importance here is that there are more than one match to a rule such that a plan actually spans a tree of possible plan executions. So instead of the GROOVE simulator that interactively allows to execute a sequence of matches, actually the GROOVE transition system generator is invoked. A plan is valid if one of the execution traces ends up in a final state.

In fact, all reachable states that can be generated given the generated plan are returned in the state space generation phase.

6.3 Example

List append problem with three rules, `next`, `append`, `return`. The plan file looks as follows

```
next
next
next
append
return
```

This simple plan (ordered list of transition applications) syntax is the one ingested by the simulator. As such plan has many possible outcomes all are reported.

By applying the sequence in the example problem two possible states are generated and stored in the files files `<prefix>-<statenr>` according the GXL-Format (cf. Appendix).

6.4 Difficulty Level

We will have some graph transformation benchmarks of rising difficulty level.

- `level_1`: *Solitaire*, *Girls' Gossiping*
- `level_2`: *List Manipulation*, *Circular Buffers*
- `level_3`: *Balanced Search Trees*
- `level_4`: *Virtual Machine*

The description of the individual domains and the corresponding graph transformation systems are made available to the competitors on the Internet. While *Girls' Gossiping*, *List Manipulation*, *Circular Buffers* exist before the competition, *Solitaire*, and *Balanced Search Trees* are made up especially for ICKEPS. The *Virtual Machine* is the most complex graph transition model and described in [8].

7 Conclusion

This paper introduces to the interconnection between planning and exploration in graph transformation systems. We used the GROOVE simulator as a case study, and showed that some traditional STRIPS planning benchmarks can be specified very intuitively. Besides Blocksworld and Logistics further successfully modeled the *Grid* and *Gripper* domain in GROOVE.

The paper also delivers first data for possible synergies opposite direction, i.e., how planner can accelerate the exploration process for graph transformation systems for a specific target graph. We have seen a PDDL description for a problem that has been identified as a challenge and shown that planners can be superior to state-of-the-art technologies.

Nonetheless the expressiveness of planners and model checkers based on graph transformation are different. On the one hand modern PDDL planners cover numerical attributes for optimizing objective functions, they allow to attach duration to rules for finding schedules of when to apply which rules to minimize the makespan. Recent additions to PDDL also cover additional soft constraints to be imposed on the set of feasible plans.

On the other hand, graph transformation systems like GROOVE feature untyped domain objects to allow the generation of a symmetry reduced state space. Two isomorphic graphs are represented only ones. Assigning names to objects, as done in planning actually destroys these symmetries. Moreover, GROOVE covers dynamic aspects such as objects and predicate creation that are not dealt with yet in the PDDL language.

We chose GROOVE as it is self-contained, platform independent, relatively stable, available to public domain and because the simulator is attached to a state space generator. Another simulators for graph transformation systems is AGG [3]. AGG features many different options such as efficient matching, consistency checking, and a critical pair analysis. An alternative for the verification of systems with graph transformation is AUGUR [10]. Recent developments in AGUR unfold and approximate the graph transformation system in form of a Petri-Net [11].

Most input formalisms for graph transformation system use XML. An overview on the status quo on the language development of GXL and GTXL can be found at `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`.

For International Knowledge Engineering Competition we summarize the important contributions of our work, which makes graph transformation an appropriate candidate for a benchmark domain.

- Graph transformation systems provide an flexible, intuitive input specification for systems of change with a sound mathematical basis. The geometric layout of the graphs associates semantics to the syntactical structures. Vied that way, simulators like GROOVE can be seen

[3] See `http://tfs.cs.tu-berlin.de/agg`

as a knowledge engineering framework for specifying planning problems. With this respect, we establish a tight connection to the KE tool *ItSimple* [15].

GROOVE has been recently extended to certain form of arithmetics on the attributes [7], so that more expressive numerical plan models can be simulated.

- Graph transformation system bridge the gap between specification and software verification as they allow to validate certain types of UML designs and software models. With the incorporation of heuristic/local search planners, bug finding as proposed with the *directed model checking paradigm* can be accelerated. A planner input encoding includes planning tool inherent guidance to the search for free, while heuristics for graph transformation systems have not yet been implemented [2]

- The main objective of this article is to provide a challenging domain for the design of plan models. The fundus for graph transformation exploration problems is continuously rising, even though no forum yet exists. Examples in GXL language (single push-out) like *list manipulation* and *buffer manipulation* come with the GROOVE distribution, while (double pushout) examples like *mutual exclusion*, *dining philosophers* and *red-black trees*, defined in GTXL can be obtained on

  `http://www.ti.inf.uni-due.de/research/augur_1/examples/index.shtml`

To steer GROOVE remotely, a file or protocol-base simulation has been provided. Using the standard notation of the international planning, this has imposed only a minor extension to the public release of GROOVE.

References

[1] *Graph Transformation. Proceedings of the 3rd International Conference.* Springer, Lecture Notes in Computer Science, Volume 4178, 2006.

[2] S. Edelkamp, S. Jabbar, and A. Lluch-Lafuente. Heuristic search for the analysis of graph transition systems. In *International Conference on Graph Transformation (ICGT)*, pages 414–429, 2006.

[3] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology*, 5(2-3):247–267, 2004.

[4] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[5] J. Hoffmann, S. Edelkamp, S. Thiebaux, R. Englert, F. Liporace, and S. Trüg. Engineering realistic benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research*, 2006.

[6] J. Hoffmann and B. Nebel. Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[7] H. Kastenberg. Towards attributed graphs in Groove. In *Proceedings of the Workshop on Graph Transformation for Verification and Concurrency*, volume 154 of *Electronic Notes in Theoretical Computer Science*.

[8] H. Kastenberg, A. Kleppe, and A. Rensink. Defining object-oriented execution semantics using graph transformations. In *FMOODS*, pages 186–201, 2006.

[9] H. Kastenberg and A. Rensink. Model checking dynamic states in GROOVE. In *Model Checking Software (SPIN)*, pages 299–305, 2006.

[10] B. König and V. Kozioura. Augur—a tool for the analysis of graph transformation systems. *EATCS Bulletin*, 87:125–137, 2005. Appeared in The Formal Specification Column.

[11] B. König and V. Kozioura. Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In *TACAS*, pages 197–211, 2006.

[12] A. Rensink. The groove simulator: A tool for state space generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, pages 479–485.

[13] A. Rensink. Towards model checking graph grammars. Technical Report DSSETR20032, Proceedings of the 3rd Workshop on Automated Verification of Critical Systems. University of Southampton, 2003.

[14] A. Rensink. Nested quantification in graph transformation rules. In *International Conference on Graph Transformation (ICGT)*, pages 1–13, 2006.

[15] T. S. Vaquero, F. Tonidande, L. N. de Barrow, and J. R. Silva. On the use of UML.P for modeling a real application as a planning problem. In *Proceedings or the 16th International Conference on Automated planning and Scheduling (ICAPS)*, pages 434–437, 2006.

[16] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In *Software Visualization*, pages 324–336. Springer, LNCS, 2002.

8 Appendix

GXL of the stack rule of the Blocksworld domain.

```
<?xml version="1.0" encoding="UTF-8"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
    <graph id="graph" role="graph" edgeids="false" edgemode="directed">
        <node id="n49"/> <node id="n50"/>   <node id="n51"/>
        <edge from="n49" to="n49">
            <attr name="label"> <string>del:clear</string> </attr>
        </edge>
        <edge from="n49" to="n49">
            <attr name="label"> <string>Block</string> </attr>
        </edge>
        <edge from="n50" to="n50">
            <attr name="label"> <string>Block</string> </attr>
        </edge>
        <edge from="n50" to="n49">
            <attr name="label"> <string>new:on</string> </attr>
        </edge>
        <edge from="n50" to="n50">
            <attr name="label"> <string>new:clear</string> </attr>
        </edge>
        <edge from="n50" to="n50">
            <attr name="label"> <string>del:holding</string> </attr>
        </edge>
        <edge from="n51" to="n51">
            <attr name="label"> <string>new:empty</string> </attr>
        </edge>
        <edge from="n51" to="n51">
            <attr name="label"> <string>Arm</string> </attr>
        </edge>
    </graph>
</gxl>
```