# Knowledge-based Graph Exploration Analysis

Ismênia Galvão[1], Eduardo Zambon[2⋆], Arend Rensink[2], Lesley Wevers[2], and Mehmet Aksit[1]

[1] Software Engineering Group,
{i.galvao, m.aksit}@ewi.utwente.nl
[2] Formal Methods and Tools Group,
Computer Science Department,
University of Twente
PO Box 217, 7500 AE, Enschede, The Netherlands
{zambon, rensink}@cs.utwente.nl, l.wevers@student.utwente.nl

**Abstract.** In a context where graph transformation is used to explore a space of possible solutions to a given problem, it is almost always necessary to inspect candidate solutions for relevant properties. This means that there is a need for a flexible mechanism to query not only graphs but also their evolution. In this paper we show how to use Prolog queries to analyse graph exploration. Queries can operate both on the level of individual graphs and on the level of the transformation steps, enabling a very powerful and flexible analysis method. This has been implemented in the graph-based verification tool GROOVE. As an application of this approach, we show how it gives rise to a competitive analysis technique in the domain of feature modelling.

**Keywords:** Graph Exploration Analysis, Prolog, GROOVE, Feature Modelling

## 1 Introduction

The practical value of graph transformation (GT) is especially determined by the fact that graphs are a very general, widely applicable mathematical structure. Virtually every artefact can be understood in terms of entities and relations between them, which makes it a graph; and consequently, changes in such an artefact can be specified through GT rules.

On the other hand, capability does not automatically imply suitability. For instance, though it is possible to express structural properties as (nested) graph conditions – see, for instance, [13, 9] – in practice, if one wants to query a given structure, writing graphical conditions to express and test for such queries is not always the most obvious or effective way to go about it. This is particularly true if the queries have not been predefined but are user-provided. Instead, there are dedicated languages suitable for querying relational structures, such as, for instance, SQL or Prolog.

⋆ The work of this author is supported by the GRAIL project, funded by NWO (Grant 612.000.632).

The need for a powerful and flexible query language becomes even more clear when one wants to combine static (structural) properties with dynamic ones, so as to include the future or past evolution of the structure. For instance, temporal logic has been especially introduced to express dynamic properties and check them efficiently (see [1] for an overview). However, besides lacking accessibility, temporal logic is *propositional*, meaning that it takes structural properties as basic building blocks; there is very little work on logics that can freely mix static and dynamic aspects of a system.

An example domain that requires this combination of static and dynamic aspects is *feature modelling*. A feature model is a graph in which nodes represent possible features (of some system under design) and edges express that one feature requires another, is in conflict, or is related in some other way. Graph transformation can be used to actually select features (in such a way that the constraints are met). The outcome is a (partially) resolved model, the quality of which is not only determined by the choices actually made but also by the possible choices still remaining. Thus, one would like to query a feature model for both its static properties (the choices actually made) and for its dynamic properties (the potential further transformation steps).

In this paper, we describe how one can use Prolog to query static and dynamic properties of graphs, simultaneously and uniformly. Besides the transformed graphs this requires a graph transition system (GTS), which is itself a graph with nodes corresponding to state graphs and edges to rule applications. The basic building block of Prolog is a *predicate*, which expresses a relation between its arguments. Example predicates in our setting are:

- The relation between a graph and its nodes or edges;
- The relation between an edge and its source or target node or its label;
- The relation between a state of the GTS and the corresponding graph;
- The relation between one state of the GTS and the next.

Using an extension of the transformation tool GROOVE that supports Prolog queries, we demonstrate the capabilities of this approach on a case study based on feature modelling.

In this paper, we first present the basic concepts for querying graphs using Prolog (Section 2); then we describe the application to feature modelling in Section 3. An analysis of the results can be found in Section 4. Conclusion and ideas for future work are given in Section 5.

## 2 Prolog in GROOVE

The Prolog programming language [5] is the *de facto* representative of the logic programming paradigm. Unlike imperative languages, Prolog is declarative: a Prolog program is composed of predicates about objects and their relations, and computations are performed by running queries over predicates. Given a

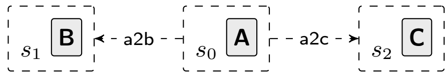


**Fig. 1.** Example GTS with three states and two transitions.

query asking whether a predicate holds for a certain (given) object, the Prolog interpreter uses a *resolution* procedure that yields a yes or no answer. On the other hand, if a query has free variables, the Prolog engine will enumerate all objects which can be assigned to the variables so as to make the predicate true. This process is called *unification*.

GROOVE [14, 8] is a graph transformation tool set which can recursively explore and collect all possible rule applications over a start graph: this is referred to as the *exploration of the state space* of a graph grammar, and produces a GTS. GROOVE has a graphical interface called the Simulator, for editing graphs and rules, and for exploring and visualising the GTS. The main technical contribution of this paper is the integration of a Prolog interpreter into the Simulator.

### 2.1 Functionality Overview

We illustrate the functionality on the basis of a very small example. Figure 1 shows a GTS with three states, represented by dashed boxes: the start state $s_0$ and two successor states $s_1$ and $s_2$. Each of the states contains a very simple graph, consisting of just a single node. The graph of $s_1$ [resp. $s_2$] is obtained from $s_0$ by applying rule a2b [resp. rule a2c] (not shown here) which replaces an **A**-node by a **B**-node [resp. a **C**-node]. Now consider the following Prolog query:

```
?- state(X), state_graph(X,GX), has_node_type(GX,'A'),
   state_next(X,Y), state_graph(Y,GY), has_node_type(GY,'C').
```

The query is composed of six predicates, interpreted conjunctively from left to right (the meaning of characters + and ? will be discussed in Section 2.2):

- state(?State) iterates over the states of the currently explored GTS.
- state_graph(+State, ?Graph) unifies the graph of the given state with the second argument; *i.e.*, it retrieves the graph associated with the given state.
- has_node_type(+Graph, +Type) succeeds if the given graph has at least one node of the given type.
- state_next(+State, ?NextState) iterates over all successors of the given state.

The purpose of the query is to search for a state (variable Y) with a graph (GY) that has at least one node of type **C** and that has a predecessor state (X) whose graph (GX) contains a node of type **A**. Running this query produces the following result, which correctly unifies Y with state $s_2$:

```
X = s0
GX = Nodes: [n0]; Edges: [n0--A-->n0]
Y = s2
```

```
GY = Nodes: [n0]; Edges: [n0--C-->n0]
Yes
More?
No
```

The output also shows the bindings for the other variables in the query. The values printed for variables GX and GY are the toString representations of the unified graphs, which show their internal structure – this explains the non-empty edge lists[3]. In the last two lines of the listing above, the user asked the interpreter if there are more results for the query. Since there are no other states that satisfy the query constraints, the answer is negative. If the GTS had more states satisfying the query, continuing the execution would eventually produce all of them. This is a consequence of the Prolog resolution procedure, which backtracks to predicate state_next, unifying Y with other successors of X, as well as to state, unifying X with other states of the GTS.

In addition to using the built-in GROOVE predicates, users can also define their own Prolog predicates. This ability to expand the Prolog knowledge-base (illustrated on Section 3.3) improves the extensibility of the framework.

### 2.2 Implementation Overview

Figure 2 shows the main elements of the integration of Prolog into the Simulator. GROOVE is written in Java, so in order to ease the coupling, we chose the GNU Prolog for Java library[4] [7] as our Prolog interpreter. The Simulator state in Figure 2 stands for the current snapshot of the Simulator configuration in memory. It contains Java objects that represent, among others, host graphs, transformation rules, and the GTS. The main block of Figure 2 is the glue code, which connects the Prolog interpreter to the rest of the Simulator. The glue code registers itself in the interpreter and is called back when a Prolog query is run. When called, the glue code inspects the Simulator state and tries to unify the Java objects with terms (variables) of the query.

**Built-in predicates.** Each built-in GROOVE predicate requires some glue code, written partly in Prolog and partly in Java. When the Prolog interpreter is created, an initialisation phase registers the built-in predicates with the interpreter. For instance, gts(−GTS) is a built-in predicate that binds the Java GTS object to a Prolog variable. Predicate registration is done with the following query:

```
:- build_in(gts/1, 'groove.prolog.builtin.Predicate_gts').
```

Predicate build_in is a special interpreter command for creating new predicates. The first argument specifies the predicate name and arity, the second one gives the name of the Java class that implements the predicate functionality. Here is a simplified listing for the Java Predicate_gts class.

[3] GROOVE uses an internal graph representation where nodes have very little structure; node types and flags are stored as special self-edges.
[4] http://www.gnu.org/software/gnuprologjava/

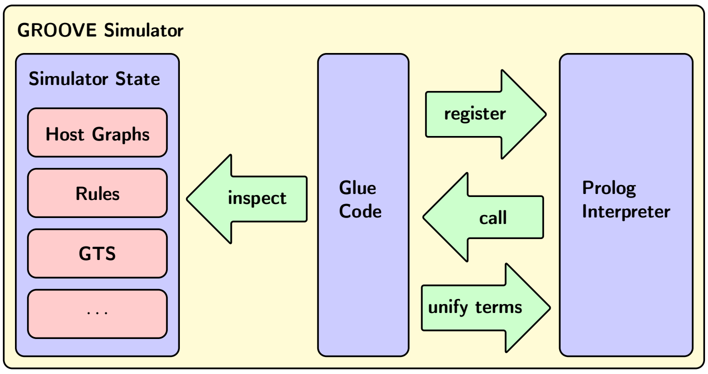


**Fig. 2.** Integration of the Prolog interpreter in the GROOVE simulator.

```
1 public class Predicate_gts extends PrologCode {
2   public int execute(Interpreter interpreter, boolean backtracking, Term[] args) {
3     GTS gts = getSimulatorState().getGTS();
4     if (gts == null) {
5       return FAIL;
6     }
7     return interpreter.unify(args[0], gts);
8   }
9 }
```

When gts(X) is evaluated in a query, the interpreter calls execute of Predicate_gts. The third argument of the method is an array of Prolog terms that corresponds to the arguments of the predicate — in this case, X. The method first inspects the Simulator state to retrieve the GTS object (line 3). If the object is **null** the query fails, otherwise the object is unified with (*i.e.*, bound to) X (line 7).

**Argument modes.** In the above, we have specified predicate signatures in which the parameter names were prefixed with special characters. These indicate the interaction of the Prolog interpreter with arguments at that position:

\+ Input parameter: the argument must already be bound to an object of the appropriate type. For example, has_node_type(+Graph,+Type) succeeds if the given graph has a node of the given type.

– Output parameter: the argument should be *free*, *i.e.*, not bound to an object; it will receive a value through the query. For example, gts(−GTS) assigns the object that represents the current GTS of the Simulator.

? Bidirectional parameter: can be used either as input or as output. For example, state(?State) may be used in two ways. If the argument is already bound to a state, the predicate either succeeds or fails depending on whether that state is part of the current GTS or not. If the argument is free, it will be unified with a state; backtracking will iterate over the remaining states.

**Backtracking.** The Prolog resolution procedure is a search for valid unifications, in the course of which it may backtrack and re-evaluate predicates to retrieve further solutions. This implies that the implementation of the built-in predicates must handle backtracking. For example, the following is the Java glue code for predicate state_next.

```
1  public class Predicate_state_next extends PrologCode {
2    public int execute(Interpreter interpreter, boolean backtracking, Term[] args) {
3      PrologCollectionIterator it;
4      if (backtracking) {
5        it = interpreter.popBacktrackInfo();
6      } else {
7        State state = getSimulatorState().getState(args[0]);
8        it = new PrologCollectionIterator(state.getNextStateSet(), args[1]);
9        interpreter.pushBacktrackInfo(it);
10     }
11     return it.nextSolution();
12   }
13 }
```

The backtracking flag (line 2) is used by the interpreter to indicate if the predicate is being evaluated for the first time in a query or if it is being called again after backtracking. During the first run, the **else** block (lines 7–9) is executed. First the state object is retrieved along with its set of successor states (call to state.getNextStateSet()). This set is put into a special iterator along with the argument to be unified (line 8), which is then passed to the interpreter and stored as backtrack information (line 9). When the method is called again during backtracking, the same iterator is retrieved from the interpreter (line 5) and the next solution is returned.

## 3 Application to Feature Modelling

Feature models [11] are commonly used to support the configuration of products in software product lines [12]. They model variability by expressing commonalities, variations and constraints between the different features that should be part of a product. A feature usually represents an aspect of the software in an early phase of the software life cycle, and the impact of the combination of features is propagated across the phases until the actual product is implemented.

The analysis of feature models [4, 16] is mostly concerned with verifying their static properties with respect to allowed specifications and valid configurations of the model. However, the specification of feature models and their configuration process go beyond the information in the model: they often involve multiple groups with distinct interests and expertise, which informally express extra properties of the features. Moreover, the definition of possible products depends on forces like market demands, user preferences, and the availability of assets at a specific time (such as the software components for the related products). Thus, feature modelling is a domain which can strongly profit from the ability to define

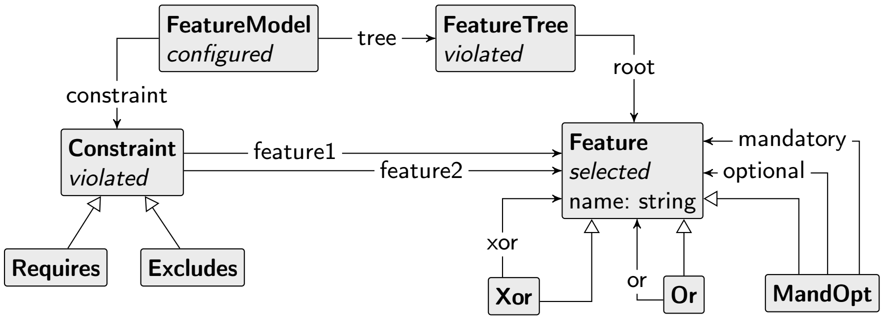


**Fig. 3.** A type graph for feature models.

and query static and dynamic properties of models, leading to richer analysis techniques. In particular, we can identify the following tasks in the analysis:

1. Model additional knowledge about features;
2. Define domain properties independently on the models, in a declarative way;
3. Simulate the configuration process;
4. Query for valid configurations with respect to conditions not expressed in the feature model;
5. Analyse alternative configuration paths and investigate the evolution of configuration stages.

We proceed to show how the Prolog extension for GROOVE can be used to implement these tasks. First we give an overview of the relevant concepts in terms of a type graph, some example rules and a small example model; then we focus on the use of Prolog to query the resulting state space of the grammar.

### 3.1 Feature Model Type Graph

Figure 3 shows a type graph for feature models (in GROOVE), based on the definitions given in [16]. The type **FeatureModel** represents a feature model composed of two parts: a **FeatureTree** whose nodes represent **Feature**s, and a set of explicit **Constraint**s between these features. The constraint **Requires** indicates that if the target node of feature1 is selected for a product, then the target node of feature2 should be selected as well. The constraint **Excludes** indicates that the target nodes of feature1 and feature2 cannot be both selected for the same product. Type **Feature** has three subtypes: **MandOpt**, **Or** and **Xor**. The edges from each of these subtypes to a **Feature** indicate which kinds of child features each subtype can have. Finally, the flags *configured*, *violated*, and *selected* in the type graph are used in GT rules to assist the configuration process and to enforce the identification of valid configurations.

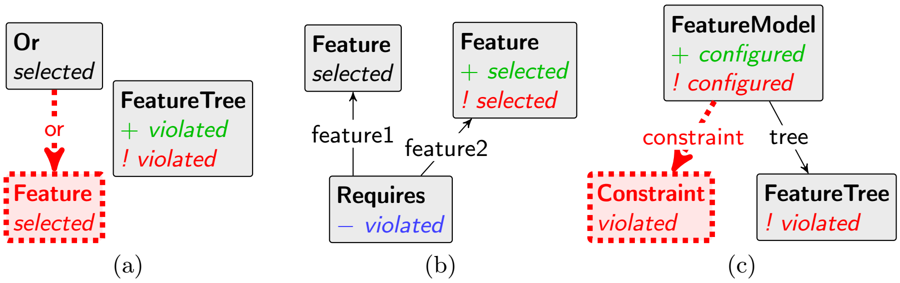


**Fig. 4.** Examples of graph production rules for feature model configuration.

### 3.2 Product Configuration

A specific feature model is a graph instantiating Figure 3, initially without any flags. The model is then configured using GT rules that encode the following requirements (some of which were discussed above):

1. The root feature must be selected first;
2. When a **MandOpt** is selected, all mandatory children must also be selected;
3. When an **Or** is selected, at least one of its children must also be selected;
4. When an **Xor** is selected, exactly one of its children must also be selected;
5. When a non-root feature is selected, its parent feature must also be selected;
6. Leaf features and intermediate child features are selected on demand.

The violation of constraints is checked at each step. This applies both to the implicit conditions of the **FeatureTree** and to the explicit **Requires** and **Excludes** constraints in the model. A valid configuration of the feature model is found when there are no violations in the tree and in the constraints. Each valid configuration selects a set of features that gives rise to a potential product of the product line.

Each of the steps above is performed by a combination of graph transformation rules. For example, Figure 4(a) shows a rule used to detect a violation on the selection of a child feature of an **Or** (step 3); the rule in Figure 4(b) selects a feature which is required by another, previously selected one and removes the violation of the **Requires** constraint; and the rule in Figure 4(c) checks the conditions for the complete feature model to be correctly configured and marks it as *configured*.

Starting from the initial feature model, the state space exploration in GROOVE generates a GTS resulting from all possible interleavings of rule applications. Each state represents the feature model with a partial selection of features, some of which may form valid configurations. Figure 5 shows a completely configured example feature model, immediately after the application of the rule shown in Figure 4(c) (named FeatureModelConfiguration in the grammar). This configuration has a set of *selected* features and no constraint violations (*violated* flags). Once generated, the GTS can be queried using Prolog.

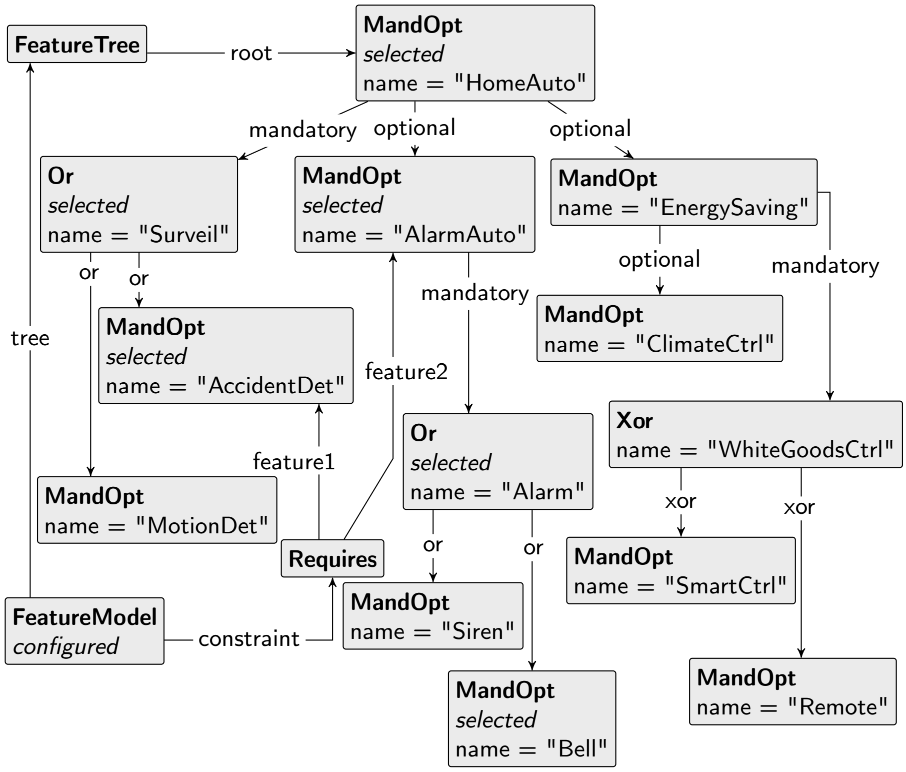


**Fig. 5.** A valid configuration of the feature model.

### 3.3 Querying the State Space

We now come to the main point of the example, which is how Prolog may be used to analyse the state space. For instance, the following user-defined predicate extracts completely configured products:

```
product(Product) :-
 rule('FeatureModelConfiguration', Rule), % Get the rule object
 % Get the graph resulting from rule application
 rule_application_result(Rule, Graph),
 % Collect all features selected to compose the product.
 findall(Feature, selected_feature(Graph, Feature), Product).
```

The predicate searches for graphs resulting from the application of rule FeatureModelConfiguration and then collects all selected features in this graph (using **findall**, which is a higher-order predicate provided by GNU Prolog). Successive calls of product generate all valid models. For the initial, unconfigured version of the feature model in Figure 5, this yields 50 products, including the one actually selected in the figure (composed of features HomeAuto, Surveil, AccidentDet, AlarmAuto, Alarm, and Bell). Predicate product uses the following auxiliary predicate, which consults information of the GTS.

```
rule_application_result(Rule, Graph) :-
 state(Source), % Get a source state
 state_transition(Source, Transition), % Get a transition from source state
 transition_event(Transition, Event), % Get the rule application event of the transition
 ruleevent_rule(Event, Rule), % Ensure that the given rule is the one that was applied
 transition_target(Transition, Target), % Get the target state of the transition
 state_graph(Target, Graph). % Get the graph of target state
```

This predicate uses the rule application event associated with transitions of the GTS to ensure that the given rule is the one that was indeed applied in the Transition.

Another useful capability provided by the Prolog extension is the possibility to define a knowledge base of additional model-related information. As an example, suppose that we are interested in products that satisfy a certain budget constraint. The following Prolog code sets the costs for each feature of the model and defines what it means for a product to be within budget.

```
% Extra facts about feature costs.
cost('HomeAuto', 1). cost('Surveil', 0). cost('WhiteGoodsCtrl', 10).
cost('AlarmAuto', 10). cost('EnergySaving', 5). cost('Siren', 15).
cost('AccidentDet', 25). cost('ClimateCtrl', 10). cost('Bell', 10).
cost('MotionDet', 25). cost('Alarm', 5). cost('SmartCtrl', 10). cost('Remote',10 ).

% Computes the cost of a product.
sum_costs([], 0).
sum_costs([H|T], Total) :- sum_costs(T, CT), cost(H, CH), Total is CH+CT.

% Checks if the given product is within the given budget.
within_budget(Budget, Product) :- sum_costs(Product, Cost), Cost =< Budget.
```

The following query returns products with total cost smaller or equal to 70:

```
?- product(P), within_budget(70, P).
```

For our running example, this gives 11 products that are within the budget constraint.

## 4 Discussion and Related Work

In the previous section we showed how the Prolog extension for GROOVE supports the graph-based representation of feature models, how extra model attributes can be specified as Prolog predicates and how state space exploration can be used to search for feature model configurations. Going back to the list of tasks in Section 3 (page 7), we see that, in fact, all of them are fulfilled.

### 4.1 Related feature modelling approaches

The analysis of feature models is useful for several reasons, such as to efficiently resolve the configuration constraints and to optimise the configuration calculation. This analysis is the object of research in many directions, which differ in the expressiveness of the models and in the configuration strategies. For example, functional programming [16], propositional formulas [3] and constraint satisfaction problems [4] have been used for the purpose of analysing feature models. Although some of these algorithms are known to be quite efficient, the big drawback of such approaches is the rigidity of the analysis method.

Extra feature attributes can be modelled in our approach in at least two ways: first, by adding new attributes to the graphs; or secondly, by defining predicates in Prolog that represent such attributes. We chose the last form because the values of the attributes used can be quite volatile in this application domain. Again, it is possible to annotate the graph with all kinds of information, but this would hamper the flexibility of the approach.

We want to draw attention to the issue of *staged configuration* of product lines. A stage corresponds to the elimination of a set of configuration options; the selection of features is deferred through stages until no variability is left. Czarnecki *et al.* [6] handle staged configuration using a feature model notation that supports the definition of feature cardinality. They explore the variability in a feature model per stage, which contains the features that can be selected. Hubaux *et al.* [10] propose a way to guide the configuration process using workflows which enforce the staged configuration in a certain order. Both approaches also handle inter-related feature models, in which the configuration order matters but is predefined and fixed over the whole configuration process.

We are able to generate all configuration stages of a feature model, as graph states, and to inspect these stages in several ways: by querying in which order the features (especially the variable features) are selected, or also by making several kinds of inspections in these stages. For example, our GROOVE solution supports the analysis of configuration contexts in which a constraint has been violated. We can also add extra constraints which are combinations of conditions in previous stages.

### 4.2 Related tools

In this section we compare the approach described in this paper to other existing combinations of graph transformation tools with Prolog. As far as we are aware, there are only two directly comparable approaches, embodied by VIATRA2 [17] and VMTS [18], both of which are GT-based tools for model transformations. For a more comprehensive comparison between GROOVE and other GT tools see [8].

**VIATRA2.** Varró and Balogh [2] describe how VIATRA2 and Prolog can be used to implement their so called Model Transformation by Example (MTBE) approach.

The purpose of MTBE is to semi-automatically derive model transformation rules from example relations between source and target model elements. These example relations are represented in VIATRA2 using a mapping model, formed by the source and target meta-models and a reference meta-model to interconnect them. The mapping model is translated to Prolog clauses and an inductive learning program is run, producing Prolog inference rules representing hypothesis that are satisfied under the given clauses. These inference rules are then translated back to a VIATRA2 representation and give rise to model transformation rules that can operate on instances of the source and target meta-models, following the example relations given in the mapping model. This process can be repeated in order to iteratively refine the rules produced.

From the above, it should be clear that the intended use of Prolog in the setting of VIATRA2 is quite different from ours, and hence there is little basis for a deeper comparison.

**VMTS.** At the GraBaTs 2009 workshop, Siroki *et al.* [15] presented a solution to the leader election case study using VMTS and Prolog.

The goal of their approach is to check if the outcome of a set of model transformation rules applied to a given input model complies to certain properties. To perform this analysis, first the input model, the transformation rules and the control flow graph specifying the order for rule applications are all translated from the VMTS format to a Prolog representation. Subsequently, the Prolog resolution procedure is used to enumerate the possible output models of the transformation. Finally, these output models are checked by Prolog predicates that express the properties one wants to assert.

Their use of Prolog resolution plays the same role as the state space exploration functionality of GROOVE. However, their approach suffers from the need to translate VMTS objects to Prolog. The Prolog resolution procedure is not adequate for the exploration of a graph-based state space and therefore gives poor performance. Another consequence of the translation is the low readability of the generated Prolog clauses.

## 5 Conclusions and Future Work

Summarising, the highlights of the approach described in this paper are:

- Prolog is tightly integrated with graph-based state space exploration;
- Queries can uniformly combine static and dynamic aspects of graphs;
- The framework supports user-defined Prolog facts and predicates.

We have demonstrated these advantages by applying the approach in the domain of feature modelling, where it gives rise to a competitive alternative to existing, more rigid frameworks.

We have implemented the above as an extension to GROOVE. Although many of the examples given in this paper could have been solved in GROOVE using other means, the Prolog-based solutions are more convenient and elegant. Therefore, the extension improves usability, which is a key factor for success.

On a more general level, this paper shows that there is much to be gained when graph transformation is connected to other techniques, and that this connection can be done in a simple, uniform way.

*Future work.* There are two main points planned as future work.

– Prolog-*based application conditions.* One can associate Prolog queries to individual GT rules, to play the role of additional application conditions. When a rule with a query is matched, the query is executed in the Prolog interpreter, and only if the query succeeds the rule is applied. This functionality is orthogonal to other application conditions already present in GROOVE, such as NACs, and would give another option for controlling the flow of rule applications, in addition to rule priorities and control programs.
– Prolog-*based state space exploration.* One can also extend the GROOVE exploration strategies with a condition based on a Prolog query. Every time a new state is produced, the query is run, and if the query is successful the state is added to the GTS. The effect is comparable to a global post-application condition.

*Availability.* The Prolog extension described in this paper is implemented in GROOVE version 4.4.0, available at `http://groove.cs.utwente.nl`. The grammar for the solution given in Section 3 can also be downloaded at the same address.

*Acknowledgement.* The integration of Prolog into GROOVE is originally due to Michiel Hendriks.

## References

1. Baier, C., Katoen, J.P.: Principles of model checking. MIT Press (2008)
2. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and System Modeling 8(3), 347–364 (2009)
3. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, J.H., Pohl, K. (eds.) Software Product Lines Conference (SPLC). Lecture Notes in Computer Science, vol. 3714, pp. 7–20. Springer Verlag (2005)
4. Benavides, D., Trinidad, P., Ruiz-Cortes, A.: Automated reasoning on feature models. In: Pastor, O., Falcao e Cunha, J. (eds.) CAiSE 2005. Lecture Notes in Computer Science, vol. 3520, pp. 491–503. Springer-Verlag Berlin Heidelberg (2005)
5. Clocksin, W.F., Mellish, C.S.: Programming in Prolog. Springer-Verlag, 2 edn. (1984)
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: Software Product Lines: Third International Conference (SPLC). pp. 266–283. Springer-Verlag (2004)
7. Diaz, D., Codognet, P.: The GNU Prolog system and its implementation. In: ACM Symposium on Applied Computing (SAC). vol. 2, pp. 728–732. ACM, New York, NY, USA (2000)

8. Ghamarian, A., de Mol, M., Rensink, A., Zambon, E., Zimakova, M.: Modelling and analysis using GROOVE. International Journal on Software Tools for Technology Transfer (STTT) (March 2011)
9. Habel, A., Pennemann, K.H., Rensink, A.: Weakest preconditions for high-level programs. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) International Conference on Graph Transformations (ICGT). Lecture Notes in Computer Science, vol. 4178, pp. 445–460. Springer (2006)
10. Hubaux, A., Classen, A., Heymans, P.: Formal modelling of feature configuration workflows. In: Muthig, D., McGregor, J.D. (eds.) 13th International Software Product Line Conference (SPLC). ACM International Conference Proceeding Series, vol. 446, pp. 221–230. ACM (2009)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
12. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
13. Rensink, A.: Representing first-order logic using graphs. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) International Conference on Graph Transformations (ICGT). Lecture Notes in Computer Science, vol. 3256, pp. 319–335. Springer Verlag, Berlin (2004)
14. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, vol. 3062, pp. 479–485. Springer (2004)
15. Siroki, L., Vajk, T., Madari, I., Mezei, G.: VMTS Solution of Case Study: Leader Election, `http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/submissions/grabats2009_submission_18-final.pdf`
16. van den Broek, P.M., Galvao, I., Noppen, J.A.R.: Elimination of constraints from feature trees. In: Proceedings of the 12th International Software Product Line Conference. vol. 2, pp. 227–232. Limerick, Ireland (2008)
17. VIATRA2 — VIsual Automated model TRAnsformations framework, `http://www.eclipse.org/gmt/VIATRA2/`
18. VMTS — Visual Modeling and Transformation System, `http://vmts.aut.bme.hu/`