Master's Thesis

Extending the Groove Control Language with Variables

Author:
Olaf Keijsers

Graduation Committee:
Dr. ir. A. Rensink
Dr. ir. R. Langerak
S. Ciraci, PhD

June 2010

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods & Tools Group

Abstract

Graphs can be used to model a variety of things, not only in computers but also in real-world settings. What we model with graphs, we would then like to check in some capacity, verifying its correctness. We do this with graph transformation tools, of which there are many and each has its own unique features. Groove is the graph transformation tool developed by the Formal Methods and Tools group of the University of Twente, and this project focuses on extending Groove with additional functionality.

Groove currently supports the scheduling of graph transformation rules using an imperative control language. In this Master's Thesis we theoretically work out a method to further control not only *when* graph transformation rules are scheduled but also *where*, by parameterising the rules in order to both get information from the host graph into the control structure as well as vice versa, telling Groove where to apply rules. We have also done the theory of letting users input constants directly from the control language, instead of having to write rules with hardcoded attributes in them. Both of these ideas have been implemented into the tool.

Using these new features, we reduce the amount of meta-information required in the rules as well as the underlying graph, instead keeping it where we think it belongs - together with the other control information. We show a number of examples which benefit from each of these new features.

Contents

Chapter 1

Introduction

1.1 Motivation

As software engineers, we do not just want to make software, we also want to somehow be sure that this software performs what it was designed to do - to verify that it meets the requirements. As software itself is often hard to test automatically, either because it is too complex or because automating tests is difficult, we use models of the software to verify the actual software. Software can be modeled in different ways using various specification methods, each with its own goals. When we have a model, we can use this model verify that it correctly implements all the requirements by using a verification method (model checking).

Graph transformation can be used as an approach to model the working of a software system, by modelling the initial state of the software as a graph and generating graph transformation rules for the various changes the software can make to its initial state. Naturally, we will want to be as expressive as possible in this, specifying rules that do exactly what we want, rather than doing several more general things and discarding the incorrect results. The Formal Methods & Tools group at the University of Twente develop a graph transformation tool called Groove, which is the tool we used for this project.

Because we generally do not want to be able to apply all of the rules we have generated on every state of the system, there is often some control structure involved, be it a simple global priority system between rules or something more complex. Groove uses an imperative control system, allowing us to write simple programs indicating which rules should be applied in which order. There is no provision in the control system however, for specifying where in the graph a rule should be applied, should several options be possible. We can get around this by having rules add meta-information to the host graph (like a special edge to indicate where a rule should apply, which can then be tested for by other rules), thus adding control information to both the host graph and the transformation rules instead of having it all in the control system.

As this approach requires the user to add logic to both the host graph (which should model the system state) and the transformation rules (which should model changes to the system state), it is suboptimal and we would like to improve it. We propose that a way to obtain information about where a rule was matched and using this information to restrict subsequent matches will help us in decoupling this functionality and moving more of the control information to the control structure. The research question we will try to answer therefore becomes *"Can we remove control information from the graph transformation rules and host graph and move it into the control program by using parameterised rules?"*

1.2 Overview

The structure of this report is as follows.

- In chapter 2, we will explain how Groove worked before this project began. We will talk about the models describing graphs, graph transformation rules, automata and the control language, and in section 2.6 we will explain some of the mathematical notations used later on in the report;

- Chapter 3 contains some of the possible extensions to the control language we developed, with examples how they could be used. We used these as a basis to decide which extensions were to be implemented;
- In chapter 4 we work out the first extension, basic input and output parameters. We show how the models presented in chapter 2 are modified to accomodate the added functionality and give examples on how this benefits the decoupling of control and transformation;
- Chapter 5 details the second extension we worked on, which adds attribute parameters (both literal and as variables) to the control language;
- In chapter 6 we explain how we implemented both of these extensions;
- In chapter 7 we show some examples where our added functionality helps in decoupling the transformation and the control information;
- Finally, in chapter 8 we discuss whether the goals we set out for ourselves have been obtained. We also have a comparison of how some of the other graph transformation tools handle control and what kind of work could be done on the Groove control language in the future to further the work we did in this project.

Chapter 2

Definitions and current state of the art

This research deals with extending Groove [12], the GRaphs for Object Oriented VErification tool developed by the Formal Methods and Tools group of the University of Twente. It consists of a set of tools to edit graphs and graph grammars as well as simulation tools to explore production systems and verify Computation Tree Logic formulas. These tools come in both graphical and command-line versions and are written in Java so as to allow them to be run on a variety of platforms. The GUI versions work on any operating system that has a Java Virtual Machine implementation and the command-line version ensures that complex state-space explorations that would exhaust the resources of a desktop PC or a laptop can be run on a computing cluster.

Added by Tom Staijen [13] is an expression language to constrain the applicability of rules beyond the mechanisms that were already in the tool before he began work on it, and to reduce unnecessary control-complexity in the rules by moving it to the control language. Extending the functionality of this language is the main focus of this research.

Note that other work has been done in the area of adding control to rule-based systems, this work will be detailed in section 8.2.

In this first chapter you will find explanations and definitions of all the concepts that are used in the rest of this report (and which need explaining). All of these concepts have been illustrated with examples.

2.1 Graphs

Groove is a graph production tool which facilitates graph transformation through the use of graph transformation rules. Before the actual transformation rules can be explained, it is necessary to detail a few key terms which are used in graph transformation. The first one of these is the concept of a graph itself.

Graphs are the main mathematical structure used in Groove. They are a very broad topic in mathematics and can be used to model many things; however, only the parts relevant to Groove and this report will be explained here.

In Groove, the graphs used are directed edge-labelled graphs. This means that they consist of nodes and edges, with the edges having a direction and only the edges having labels. While nodes can not have labels, self-edges (an edge with equal source and target) are generally displayed as node-labels for convenience. We denote the set of all possible labels by Labels.

Definition 2.1.1 (graph) *A graph is a tuple $G = (V, E)$ where*

- *V is a set of vertices;*
- *$E : V \times \mathsf{Labels} \times V$ is a set of edges.*

We use V_G as the set of vertices of G and E_G as the set of edges of G.

Example 2.1.1 *An example graph G_1, can be found in figure 2.1. This graph depicts (part of) the state of an ant hill simulation called AntWorld [20], with a "home" (the* AntHill *node) and two places ants could visit to look for food (neither actually has food on it at this time). It also*

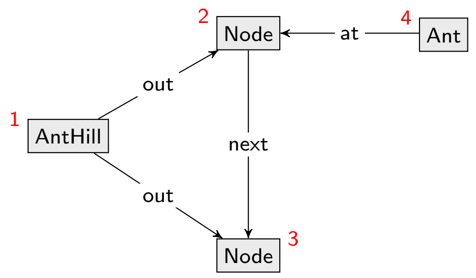


Figure 2.1: An example graph

contains one ant which is at one of the two places. Note that the numbers next to the nodes are not part of the graph, these can be considered the "internal" numbering of the nodes (without them we would have no way to identify the vertices in the formal definition). This graph is formally defined as:

$$\begin{aligned}
G_1 &= (V_1, E_1, L_1)\\
V_1 &= \{1, 2, 3, 4\}\\
L_1 &= \{\text{Ant}, \text{Anthill}, \text{at}, \text{next}, \text{Node}, \text{out}\}\\
E_1 &= \{(1, \text{Anthill}, 1), (1, \text{out}, 2), (1, \text{out}, 3), (2, \text{Node}, 2), (2, \text{next}, 3),\\
&\qquad (3, \text{Node}, 3), (4, \text{Ant}, 4), (4, \text{at}, 3)\}
\end{aligned}$$

2.2 Morphisms

As graph transformation rules are a combination of graphs and morphisms, we will now explain what a morphism is, along with an example.

In mathematics, a morphism is a mapping between two structures, which in this report will be graphs. A morphism f is defined on its domain (or source) X and codomain (or target) Y.

In graphs a morphism is a tuple of two functions:

Definition 2.2.1 (graph morphism) *A morphism f over two graphs G_1 and G_2 is a tuple (f_V, f_E) where:*

- $f_V : V_{G_1} \to V_{G_2}$ *maps the vertices in G_1 to the vertices in G_2;*
- $f_E : E_{G_1} \to E_{G_2}$ *maps the edges in G_1 to the edges in G_2.*
- $\{u, v\} \in E_{G_1} \implies \{f_V(u), f_V(v)\} \in E_{G_2}$

We write $f : G_1 \to G_2$.

We speak of a *full morphism* when for every element in X there is a corresponding element in Y (note that several elements in X may still map to the same element in Y). If this is not the case, we are dealing with a *partial morphism*. Both are used in the application of graph transformation rules.

Example 2.2.1 *An example of a morphism is provided in figure 2.2. Note that several other morphisms also exist between the left and the right graphs, as there are for instance two nodes labelled 'ant' in the right graph. As the middle node in the left graph does not have a label, it can be mapped to any node that adheres to the other requirements (has an incoming edge with label 'at' and an outgoing edge with label 'out').*

Isomorphism An isomorphism is a special morphism in which each vertex and edge in the domain has a bijective image in the codomain and vice versa. Thus an isomorphism between two

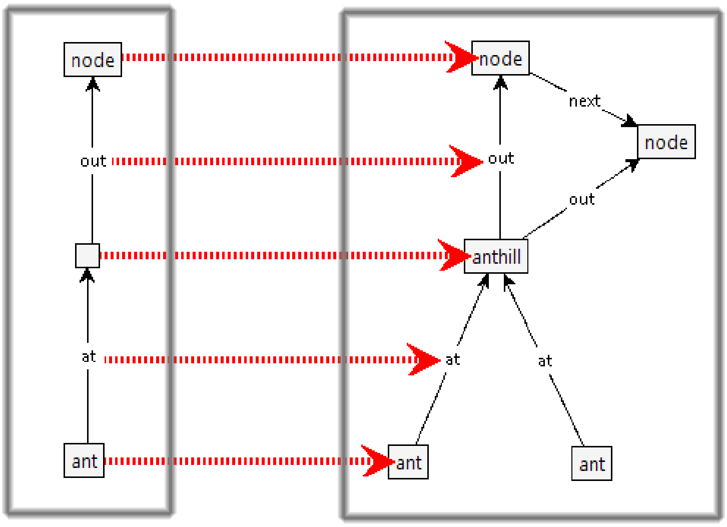


Figure 2.2: An example morphism, the dashed red lines indicate mappings.

graphs means that they are the same 'looking' graph, laid out differently. This notion is used in Groove to reduce the state space by not exploring isomorphic graphs more than once. Formally, an isomorphism is a morphism $f : X \to Y$ for which $f^{-1} \circ f = id_X$ and $f \circ f^{-1} = id_Y$ (where id is the identity function).

2.3 Graph transformation rules

Groove's functionality is centered around having one or more graph rules which can be applied to modify a graph. The combination of all of the rules is usually called a *graph rewriting system.*

We say that a rule is *scheduled* (regardless of whether it is actually applicable) when in the current system state this rule is allowed to be matched. By default, Groove will schedule all rules in every state.

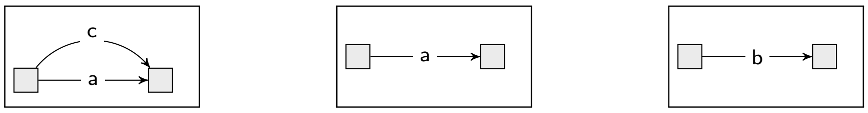
Definition 2.3.1 (graph transformation rule) *A graph transformation rule P is a tuple $(LHS, RHS, r, NACs)$ in which:*

- *LHS is a graph representing the left-hand side of the rule;*
- *RHS is a graph representing the right-hand side of the rule;*
- *$r : LHS \to RHS$ is a partial morphism (called the rule morphism) from LHS to RHS;*
- *$NACs$ is a set of tuples (NAC, n) in which:*
  - *NAC is a graph representing a negative application condition;*
  - *$n : LHS \to NAC$ is a morphism from LHS to NAC.*

The rule can be applied to a host graph G if and only if:

- $\exists f : LHS \to G$
- $\forall (NAC, n) \in NACs : \nexists g : f = g \circ n$

As this research only concerns itself with restricting which rules are applicable at which times and not with how rules are applied, a description of the application of a rule is beyond the scope of this report, but a detailed description can be found in for instance [3]. An example rule application is presented on page 7.



(a) NAC (b) LHS (c) RHS

Figure 2.3: A very simple graph transformation rule

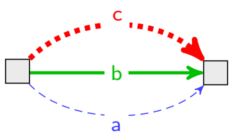


Figure 2.4: The Groove representation of the rule in figure 2.3

Example 2.3.1 *Consider a rule that looks for two nodes with an a-labelled edge between them, which do not have a c-labelled edge between them. It then deletes the a-labelled edge and creates a new b-labelled edge. The graphical representation of this can be found in figure 2.3.*

2.3.1 Graph rules in Groove

In Groove, graph transformation rules are depicted a little differently. Rather than having a separate LHS, RHS and NAC view, all these are combined into one graph with different color markings. These markings are as follows:

- Black nodes and edges are part of the LHS and must be matched for the rule to be applicable.
- Blue (dashed) nodes and edges are also part of the LHS, but will be removed by applying the rule. As such, in the normal representation these would be present in the LHS but not in the RHS.
- Green (bold) nodes and edges are newly created by applying the rule and as such are present in the RHS but not in the LHS.
- Red (bold dashed) nodes and edges are part of a NAC and must not be matched for the rule to be applicable.

Figure 2.4 shows the same rule as in figure 2.3 using Groove's notation.

Attributes A more complex rule can be found in figure 2.5. This illustrates a rule from the AntWorld case[1], which depicts the situation where an ant can move to a field one level further away from the anthill (ants start at the center and every node further away from the center is connected through an edge with the *out* label) if that field contains food. As before, the blue dashed edge will be deleted and replaced with the green edge. The new elements mean the following:

- The round node without a label is an attribute node, which means it can have any attribute value in the host graph (though the way it is used in conjunction with an integer and the *eq* operation implies that it must be an integer itself in order for this rule to match). It denotes the amount of food currently on the field.
- The diamond node is a product node, which is a tuple of data nodes.
- The oval node with the 0 label is an integer literal.

[1]AntWorld[20] is a case study designed as benchmark. It defines some simple rules which ants use to traverse a grid in search for food, returning and leaving pheromones behind when they find food.

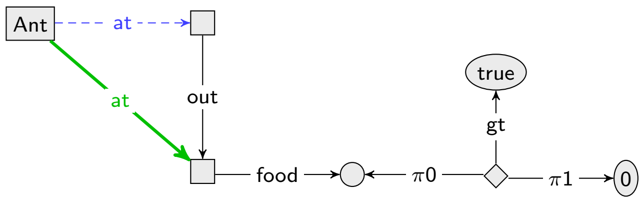


Figure 2.5: A rule from the Antworld case in Groove notation

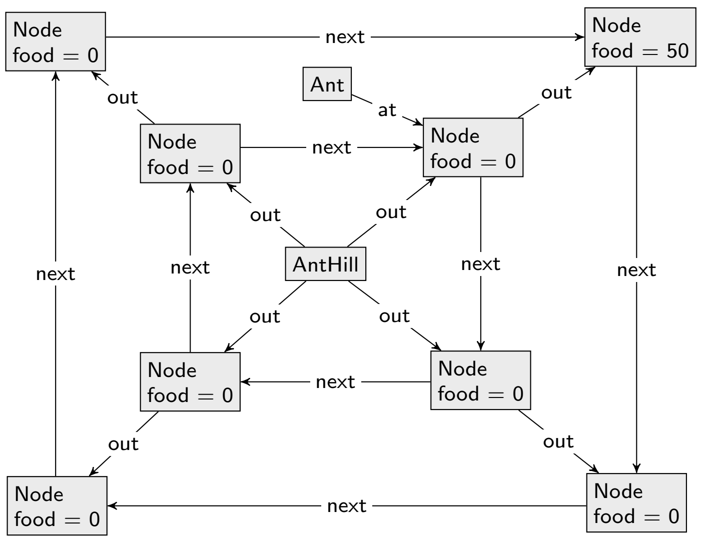


Figure 2.6: An example graph to which the rule in figure 2.5 can be applied

- The two edges labelled $\pi 0$ and $\pi 1$ are the arguments to the product node.
- The edge labelled *gt* is the operator edge for the product node, which selects which operator will be used. In this case *gt* means that it will check whether the first argument is greater than the second.
- The oval node labelled *true* is a boolean literal, which indicates that the result of the product node should be true.

An example result of the application of this rule to the graph in figure 2.6 is in figure 2.7.

Quantified rules Rules in Groove can also contain quantified nodes, indicating that the transformation this rule specifies should be applied to all nodes that match. There is also functionality in place to specify that a node must be matched at most once or at least once. The advantage of this is that things that are often repeated many times for a number of nodes can be combined into one rule-application, reducing the LTS; although in some cases it allows us to express things that were otherwise not possible, see for example [15].

Example 2.3.2 *Figure 2.8 shows a rule using a quantified node (∀). This rule specifies that for each* `Node` *in the host graph, the* `pheromones` *attribute should be multiplied by* 0.9*, indicating that pheromones left behind by an ant should decay over time. Had we done this without quantification, we would have many rule applications, one for each node. Additionally, we would have to keep track of which nodes had their pheromones reduced already and which have not.*

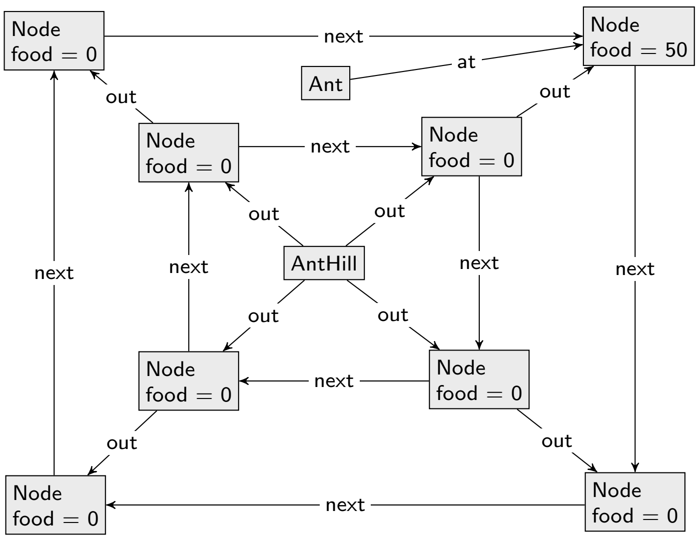


Figure 2.7: The result of the application of the rule in figure 2.5 to the graph in figure 2.6

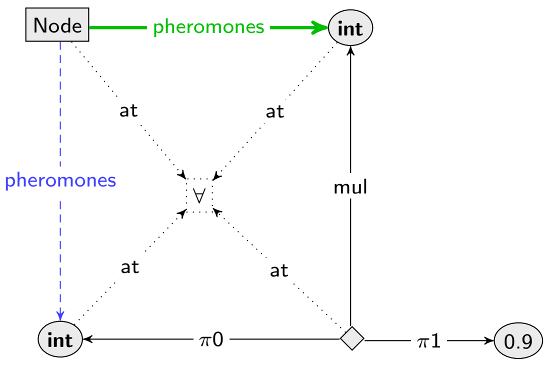


Figure 2.8: An example rule using a quantified node

2.4 Automata

When we combine the rule system with a start graph, we can generate a transition system which, when fully explored, will have all the possible reachable transformed versions of the start graph after scheduling every rule in every state (note that this transition system might be infinite in size, in which case Groove can not generate the full state space without applying some restrictions first). This type of transition system, with the states being graphs and the transitions having labels indicating which rule is applied and where, is called a *system automaton*. In order to explain this term, we will first define a general automaton.

Definition 2.4.1 (automaton) *An automaton* $\mathcal{A}$ *is a tuple* $(Q, \Sigma, \rightarrow, q_0, S)$ *where:*

- Q *is a finite set of states;*
- Σ *is a finite alphabet, which may include the special symbol* λ*;*
- $\rightarrow \subseteq Q \times \Sigma \times Q$ *is a set of transitions;*
- $q_0 \in Q$ *is the start state;*
- $S \subseteq Q$ *is a set of success states.*

Transitions can be labelled with λ to indicate that they are internal transitions, e.g. transitions which perform no observable action. $\mathcal{A}$ is called deterministic *if for all $q \in Q$ and $\ell \in \Sigma$, $q \xrightarrow{\ell} q_1$ and $q \xrightarrow{\ell} q_2$ implies $q_1 = q_2$ and $\ell \neq \lambda$.*

System automata A system automaton is a specific automaton in which every edge has a rule name and an application identifier (the match morphism) as its label. The states represent graphs, and any transition from a state G to another state G', denoted $G \xrightarrow{P,i} G'$ means that the underlying graph can G from the first state can be transformed into the underlying graph G' of the target state by applying rule P using application identifier i as the match morphism. By including these application identifiers, we artificially remove any possible nondeterminism (see the paragraph on determinism below for a description) from the system automaton, while still keeping it visible in the automaton (the transitions can still have the same rule name, but the combination of a rule name and application identifier will always be unique for a given state). Thus, we can still check the properties we want to check in the nondeterministic system automaton. The system automaton should only have nondeterminism if this was in the modelled system. If it has such nondeterminism, hiding it would change the meaning of the system automaton.

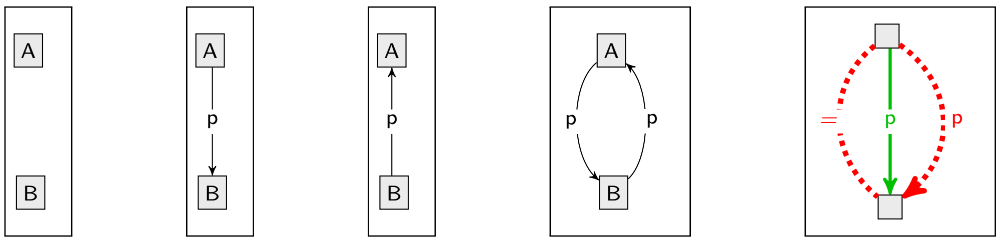
We use Id to denote the set of all possible application identifiers and Rule to denote the collection of rules.

Definition 2.4.2 (system automaton) *A* system automaton *is an automaton with $\Sigma = (\mathsf{Rule} \times \mathsf{Id}) \cup \{\lambda\}$, such that every $q \in Q$ has an associated graph $d_q \in \mathsf{Data}$, satisfying the following consistency properties:*

$$\begin{aligned} q \xrightarrow{\lambda} q' \quad &:\Leftrightarrow \quad d_q = d_{q'} \\ q \xrightarrow{r,i} q' \quad &:\Leftrightarrow \quad d_q \xrightarrow{r,i} d_{q'} \quad . \end{aligned}$$

Determinism Determinism in the context of automata means that for every state, given an observation of inputs, one transition and subsequently one target state can be determined. Because in Groove a rule can possibly be matched on several locations on a single graph, the resulting transition system would be nondeterministic. To counter this, we include the match morphism in the transitions. The combination of a rulename and a morphism makes the transitions outgoing from a single state unique and as such the system automaton will be deterministic.

Example 2.4.1 *Consider the start graph G_1 in figure 2.9a, with the very simple rule P (which adds a p-labelled edge between any two distinct nodes (the NAC-edge with the equals sign denotes that the two nodes cannot be the same) if there is no such edge already) from figure 2.9e. It is apparent that this rule can be applied to G in two ways through transitions $G \xrightarrow{P,1} G_1$ and $G \xrightarrow{P,2} G_3$, both resulting in a different graph. These two graphs are shown in figures 2.9b and 2.9c respectively. The resulting system automaton from this is one where the start state represents G_1 and has two successor states, G_2 and G_3. After the rule has been applied it can be applied again, which ends us in state G_4 (figure 2.9d). The system automaton is shown in figure 2.10.*



(a) G_1 (b) G_2 (c) G_3 (d) G_4 (e) P

Figure 2.9: The four states of the example system automaton and the production rule P

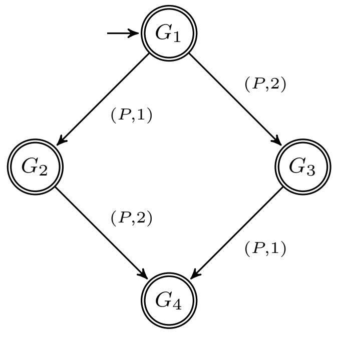

$$\begin{tikzcd}
 & G_1 \arrow[dl, "{(P,1)}"'] \arrow[dr, "{(P,2)}"] & \\
G_2 \arrow[dr, "{(P,2)}"'] & & G_3 \arrow[dl, "{(P,1)}"] \\
 & G_4 &
\end{tikzcd}$$

Figure 2.10: The system automaton generated with the start graph G_1 and rule P

Rule priorities Note that a system automaton is obtained from the rule system and start graph without control over which rules are scheduled when - all rules are scheduled in every state. In order to limit the amount of rules that are scheduled in each state, we can assign a *priority* to each rule. When we assign priorities, Groove will divide the rules into groups of the same priority. It will then for each state first schedule the rules of the highest priority group, and check if any of these rules match in that state. If no rule from a priority group has a match, it will schedule all the rules from the priority group below it and it will keep doing so until at least one rule from a priority group matches, or no more priority groups exist (in which case the state is considered a final state).

2.5 Control language in Groove

In order to apply more control as to which rules are scheduled than just using priorities (or no control at all), Tom Staijen [13] has built a mechanism for Groove which allows us to schedule specific rules using a simple expression language. We will now review the state of this control language as it was when this project started. The syntax of the control language can be found in figure 2.11. Note that both this form of control as well as the priorities explained in the previous section can also be expressed through a normal grammar without these augmentations. However, in many cases this is highly non-trivial and probably not a good idea. Groove without control is Turing-complete, but implementing even simple control in a graph grammar requires a lot of effort. The added forms of control are a convenience which allows us to concisely describe the intended behaviour rather than write rules with really large NAC portions to make sure other rules cannot be applied.

The expressions written in this language are transformed into a *control automaton.* Groove then takes the product of this control automaton with the system automaton produced by the grammar without any control to generate the product automaton, which will be used to “run” the grammar.

An example problem modelled with and without control will now be presented. Note that this example is quite trivial but should still show how control can move complexity from the rules to an expression. The example is a very small subset of what a game of Ludo [18] would look like. We have modelled only which actions a player takes in his turn in order to show that this sort of complexity can be described either by the rules or by the control language.

The no-control version of this has three rules and a starting graph (figure 2.12):

- The `throw_die` rule (figure 2.13a) will let the player with the *current*-labelled self edge throw the dice (not modelled here) and then move the *at*-labelled edge to the ‘Move pawn’ node, if the current player has an *at*-labelled edge to the ‘Throw die’ node;
- the `move_pawn` rule (figure 2.13b) will let the current player move his pawn (also not modelled here) if she has an *at*-labelled edge to the ‘Move pawn’ node, and will then remove this edge as there are no more actions to perform this turn;

- rule schedules the execution of a single rule (named rule);
- **true** behaves like a rule that is always successful and does not change the underlying structure;
- $P_1|P_2$ is the *non-deterministic choice* of P_1 and P_2;
- $P_1; P_2$ is the *sequential composition* of P_1 and P_2;
- $P*$ (the *Kleene closure*) schedules P an arbitrary number of times;
- $P+$ schedules P one or more times (short for $P; P*$);
- **alap** P (*as long as possible*) schedules P until it fails;
- **try** P_1 **else** P_2 schedules P_1 first, and schedules P_2 in case P_1 fails;
- **function** $F\{P\}$ declares a function F with body P.
- f, where f is the name of a function calls that function, effectively inlining its body in the control program.

Figure 2.11: The syntax of the control language

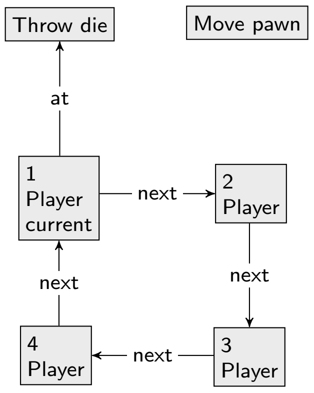
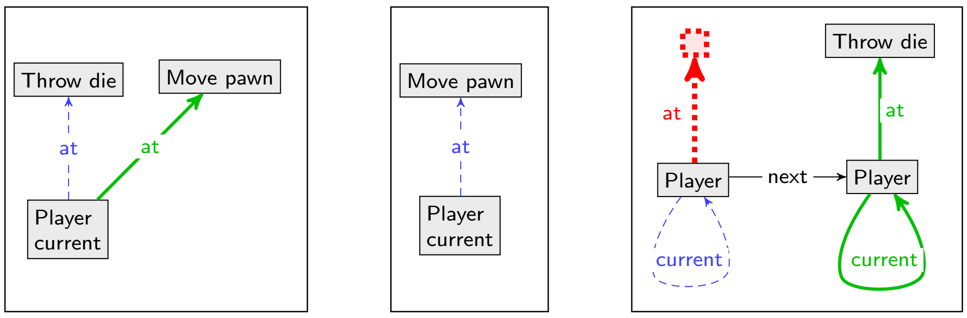


Figure 2.12: The starting graph for the no-control version of Ludo

- the `next_player` rule (figure 2.13c) will remove the *current*-labelled self-edge from the player and add it to the next player in turn.



(a) throw_die (b) move_pawn (c) next_player

Figure 2.13: The rules in the no-control version of Ludo

In this way of modelling the game, control information is in the host graph as well as in the rules, while the rules should only be concerned with *how* a move is performed (e.g. generating

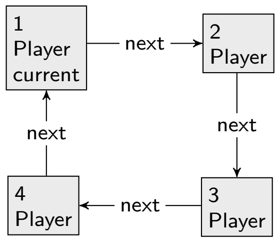


Figure 2.14: The starting graph for the control version of Ludo

a random number, choosing which pawn to move and moving it, removing any other pawns on the target location, et cetera). If we add a control program such as the one in listing 2.1, we can trivialize the `throw_die` and `move_pawn` rules (this is because not everything has been modelled. In the real model of the game, these rules would be much larger but the control information would still be removed). The `next_player` rule would lose the NAC that states there cannot be a player who is doing something. The new start graph would be as in figure 2.14.

```
alap {
  throw_dice;
  move_pawn;
  next_player;
}
```

Listing 2.1: Ludo control program

The control automaton can express more than just the application of a rule. It can also express the observation that a certain set of rules is *not* applicable, called a *failure* caused by optional rules in the control expression, such as `try{a;}`. In this case, if a is applicable it will be applied, if not the system will still continue while observing that a is not applicable. Note that for a state, outgoing edges with a failure set can only contain rule names that are on outgoing edges for normal transitions too (in other words, we cannot test for the non-applicability of rules that have no transition if they are applicable).

We use $\texttt{Fail} = 2^{\mathsf{Rule}}$ to denote the set of all possible failure sets. Failures are denoted by enclosing them in square brackets.

Definition 2.5.1 (control automaton) *A* control automaton *is an automaton where* $\Sigma = \mathsf{Rule} \cup \texttt{Fail}$*, such that:*

- *For all* $q \in Q$, $q \xrightarrow{F}$ *with* $F \in \texttt{Fail}$ *only if* $\forall a \in F : q \xrightarrow{a}$.
- *For all* $q \in S$, $a \in \mathsf{Rule}$, $F \in \texttt{Fail}$ *there is no transition* $q \xrightarrow{a}$ *or* $q \xrightarrow{F}$.

We use $[\![\mathcal{E}]\!]$ *to indicate the control automaton generated from a control expression* $\mathcal{E}$.

The latter criterion indicates that there are no outgoing transitions from a final state, which makes the composition easier.

Example 2.5.1 *We present an example of a simple control program with the associated control automaton. The control program is in listing 2.2, with the corresponding control automaton in figure 2.15.*

```
try { rule_one; } else { rule_two; }
alap { rule_three; }
rule_four;
```

Listing 2.2: Example control program

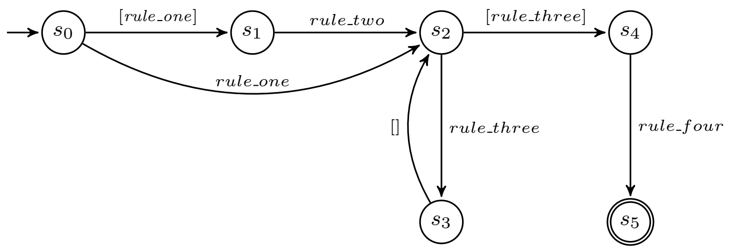


Figure 2.15: The control automaton corresponding to the control program shown in listing 2.2

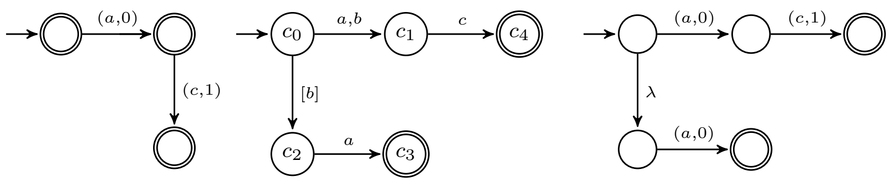


Figure 2.16: Example of a nondeterministic product where the two inputs are deterministic

2.5.1 Product of system automaton and control automaton

To generate the product of a system automaton and a control automaton, Tom Staijen has defined a new automaton whose states are tuples of states of the two original automata. The result is an automaton like the system automaton, but constrained by the control automaton on which rules can be applied in which states.

Definition 2.5.2 (product) *The* product *of a system automaton $\mathcal{A}$ and a control automaton $\mathcal{C}$ is defined as $\mathcal{A} \times \mathcal{C} = (Q_{\mathcal{A}} \times Q_{\mathcal{C}}, \Sigma_{\mathcal{A}}, \rightarrow, (q_{0,\mathcal{A}}, q_{0,\mathcal{C}}), S_{\mathcal{A}} \times S_{\mathcal{C}})$, where the transition relation is defined by the following rules:*

$$\frac{q_{\mathcal{A}} \xrightarrow{n,i}_{\mathcal{A}} q'_{\mathcal{A}} \quad q_{\mathcal{C}} \xrightarrow{n}_{\mathcal{C}} q'_{\mathcal{C}}}{(q_{\mathcal{A}}, q_{\mathcal{C}}) \xrightarrow{n,i} (q_{\mathcal{A}}, q'_{\mathcal{C}})} \qquad \frac{q_{\mathcal{C}} \xrightarrow{F}_{\mathcal{C}} q'_{\mathcal{C}} \quad \forall n \in F, k \in \mathtt{Id} : q_{\mathcal{A}} \not\xrightarrow{n,k}_{\mathcal{A}}}{(q_{\mathcal{A}}, q_{\mathcal{C}}) \xrightarrow{\lambda} (q_{\mathcal{A}}, q'_{\mathcal{C}})}$$

$$\frac{q_{\mathcal{A}} \xrightarrow{\lambda}_{\mathcal{A}} q'_{\mathcal{A}}}{(q_{\mathcal{A}}, q_{\mathcal{C}}) \xrightarrow{\lambda} (q'_{\mathcal{A}}, q_{\mathcal{C}})}$$

2.5.2 Determinism

The resulting automaton from this product can be nondeterministic, even if both the system automaton and the control automaton are deterministic. An example of this can be seen in figure 2.16. The system automaton on the left and the control automaton in the middle (corresponding to the control expression `choice {try{a; c;}}or{try{b; c;}else{a;}}`) are both deterministic (note that the observation of $[b]$ is *not* a λ-transition), while the product automaton on the right is not; the rule application $(a, 0)$ occurs twice, after which on one branch the rule application $(c, 1)$ may occur and on the other it may not.

Introducing this kind of nondeterminism in the product automaton is not a desirable effect, due to the fact that the control automaton is meant to restrict the system automaton rather than introduce nondeterminism that was not there in the first place. While nondeterminism in the system automaton (and possibly even the product automaton) is fine when it is present in the modelled system, introducing it due to the way the automaton is constructed is not. Thus we adapt the control automaton into a *guarded control automaton*. The product of a system automaton with a guarded control automaton is always deterministic[13].

Even though a determinized automaton is generally (much) larger than its nondeterministic counterpart, we choose to determinize the control automaton as removing nondeterminism from the product automaton tends to be a lot more work; control expressions are generally quite simple and as such their corresponding automata will be relatively small (< 100 states), while product automata can be many orders of magnitude larger.

In this automaton, every transition consists of a rule name and two sets of rules, one of which acts as a negative guard and one as a positive guard. A transition is only enabled if all rules in the positive guard are applicable and all rules in the negative guard are not. We use $q \xrightarrow{[F|A]n} q'$ to denote a transition with guards, where A is the positive guard, F is the negative guard and n is the rule that will be applied to get from q to q'. To denote an empty positive guard we use $q \xrightarrow{[F]n} q'$ and when both guards are empty we use $q \xrightarrow{n} q'$.

Definition 2.5.3 (guarded control automaton) *A guarded control automaton is an automaton with* $\Sigma = \texttt{Fail} \times 2^{\mathsf{Rule}} \times \mathsf{Rule}$ *and* $S \subseteq Q \times \texttt{Fail}$. *Transitions should satisfy the following constraints for all* $q \in Q$*:*

1. $q \xrightarrow{[F|A]n}$ *implies* $F \cap A = \emptyset$
2. $q \xrightarrow{[F_1|A_1]n}$ *and* $q \xrightarrow{[F_2|A_2]n}$ *implies* $F_1 \cup A_1 = F_2 \cup A_2$
3. $q \xrightarrow{[F|A]n}$ *implies* $\exists q \xrightarrow{[F \cup A]n}$.

Note that success states are now also guarded, meaning that a success state has an optional set of rules that must fail before we consider it to be an actual success state. We define a way to generate this guarded control automaton from a normal control automaton; for this we will need the failure dependency function fd. This function returns the union of all possible failures that lead to a state where n is allowed.

Definition 2.5.4 (failure dependency) *The failure dependency in a set of states qs for a rule n is defined by:*

$$fd(qs, n) = \bigcup \{F_i \mid \exists q \in qs : q \xrightarrow{F_1 \ldots F_n} q' \xrightarrow{n}\}$$

The *determinisation* of the control automaton is then given by a function *det*:

Definition 2.5.5 (control automaton determinisation) *Given a control automaton* $\mathcal{C}$, $\det(\mathcal{C})$ *is an automaton with* $Q = 2^{Q_\mathcal{C}} \setminus \emptyset$, $q_0 = \{q_{0,\mathcal{C}}\}$,

$$S = \{(q, \cup_i F_i) \mid \exists q_\mathcal{C} \in q : q_\mathcal{C} \xrightarrow{F_1 \ldots F_n} q'_\mathcal{C} \in S_\mathcal{C}\},$$

and $\rightarrow$ *is defined by:*

$$\frac{F \subseteq fd(q,n) \quad A = fd(q,n) \setminus F}{q \xrightarrow{[A|F]n} \{q'_\mathcal{C} \mid q_\mathcal{C} \in q \xrightarrow{F_1 \ldots F_n} \xrightarrow{n} q'_\mathcal{C}, F_1 \cup \ldots \cup F_n \subseteq F\}}$$

The states in the guarded control automaton are sets of states from the original control automaton. The target state of a transition $q \xrightarrow{[F|A]n} q'$ is the set of states that could be reached in the original automaton with the failures in F followed by a rule named n. The positive guard A contains the rules that are not in F but are in $fd(q, n)$. A success state is a set of tuples consisting of states and failures (these failures can be empty, in which case the success state is unconditional).

An example of a guarded control automaton can be found in figure 2.17. This is the determinized version of the control automaton in the middle part of figure 2.16.

In order to combine a guarded control automaton with a system automaton to form a new product automaton, we define the guarded product. In this definition, the function *enabled* is a function which returns the set of rules which are enabled in a system state $q_\mathcal{A}$ or any state reachable from $q_\mathcal{A}$ after any number of λ's.

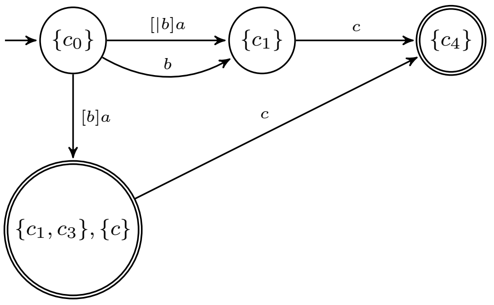


Figure 2.17: Determinized version of the control automaton from figure 2.16

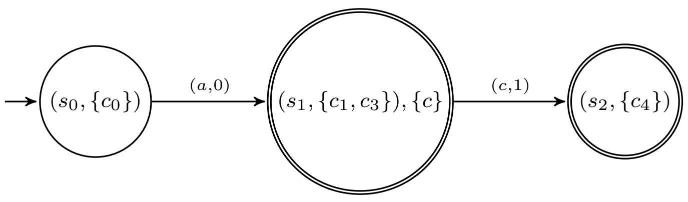


Figure 2.18: Example of a guarded product

Definition 2.5.6 (enabled rules) *Given a system automaton $\mathcal{A}$, the function enabled* $: Q_\mathcal{A} \to 2^{\mathsf{Rule}}$ *is defined as:*

$$enabled(q_\mathcal{A}) = \{n \in \mathsf{Rule} \mid \exists i \in \mathsf{Id} : q_\mathcal{A} \xRightarrow{(n,i)}_\mathcal{A}\}$$

Definition 2.5.7 (guarded product) *Given a system automaton $\mathcal{A}$ and a guarded control automaton $\mathcal{G}$, the product $\mathcal{A} \times \mathcal{G}$ is a system automaton, with $Q \subseteq Q_\mathcal{A} \times Q_\mathcal{G}$, $q_0 = (q_{0,\mathcal{A}}, q_{0,\mathcal{G}})$, and $\to$, S are defined by:*

$$\frac{q_\mathcal{A} \xrightarrow{(n,i)}_\mathcal{A} q'_\mathcal{A} \quad q_\mathcal{G} \xrightarrow{[F|A]n}_\mathcal{G} q'_\mathcal{G} \quad F \cap enabled(q_\mathcal{A}) = \emptyset \quad A \subseteq enabled(q_\mathcal{A})}{(q_\mathcal{A}, q_\mathcal{G}) \xrightarrow{(n,i)} (q'_\mathcal{A}, q'_\mathcal{G})}$$

$$\frac{q_\mathcal{A} \xrightarrow{\lambda}_\mathcal{A} q'_\mathcal{A}}{(q_\mathcal{A}, q_\mathcal{G}) \xrightarrow{\lambda} (q'_\mathcal{A}, q_\mathcal{G})} \qquad \frac{q_\mathcal{A} \in S_\mathcal{A} \quad (q_\mathcal{G}, F) \in S_\mathcal{G} \quad F \cap enabled(q_\mathcal{A}) = \emptyset}{(q_\mathcal{A}, q_\mathcal{G}) \in S}$$

A transition with rule n in $\mathcal{G}$ can 'synchronize' with rule applications in $\mathcal{A}$ when none of the rules in F are applicable in $q_\mathcal{A}$ and all of the rules in A are applicable in $q_\mathcal{A}$.

Proposition 2.5.8 (guarded product automaton determinism) *Given a guarded control automaton $\mathcal{G}$and a deterministic system automaton $\mathcal{A}$, $\mathcal{G} \times \mathcal{A}$ is deterministic.*

As an example, the product of the guarded control automaton from figure 2.17 with the system automaton on the left side of figure 2.16 is in figure 2.18.

2.6 Mathematical notations

This section contains explanations of some uncommon notations used in this report.

2.6.1 Function composition

A composed function is a function which is obtained by applying two functions in succession. The notation for this is $f = g \circ h$, which means f is obtained by applying g to h. In a more concrete example, $V \circ iVars : \mathsf{idx} \to N$ is a function mapping indices to nodes, constructed by applying $V : \mathsf{Var} \to N$ (which is a valuation function mapping variables to nodes) to $iVars : \mathsf{idx} \to \mathsf{Var}$ (which maps indices to variables).

2.6.2 Function overriding

We can restrict a function f by limiting its domain to d by writing $f\,|_d$. This is used for the following notation:

In order to override certain valuations of a function (and to add new valuations to it), we use a notation of the form $v' = v[w]$. What this means is that we let v' be all the mappings from v after removing $dom(w)$ from $dom(v)$; we then add all mappings in w to v'. In short: $v' = v\,|_{dom(v)\setminus dom(w)} \cup w = v[w]$.

2.6.3 Subfunction

We use the notation $f \sqsubseteq g$ to denote that f is a partial subfunction of g. This holds if:

1. $dom(f) \subseteq dom(g)$ and
2. $\forall x \in dom(f) : f(x) = g(x)$

Chapter 3

Possible extensions to the control language

This chapter details some of the extensions to the control language we have thought of that could benefit the functionality of the Groove control language. Not all of these are equally useful, and we will explain the advantages and disadvantages of each extension here.

While the control language already takes some work and complexity away from the graphs and rules, several extensions can be conceived that will further this effort. While all of them will have to be used carefully, in the right place these extensions would benefit rule makers in being able to write their rules more concisely and putting the complexity where it belongs (note that this location is not always the control language). Thus, the "requirements" for an extension are as follows:

1. reduce the complexity of rules and the amount of data required in the host graph which would normally be used to limit where rules can be applied;
2. better separation of concerns. Control information does not belong in the host graph and is better left to the control program;
3. reduce the state space for the combined system and control automaton thus allowing for faster verification and/or verification of bigger systems;
4. maintain backward compatibility. Existing grammars should still work with the extension in place.

While the last requirement could result in improved speed for Groove since fewer matchings have to be considered and it provides a starting point when looking for a match, the former two are what we are really after in this research. Being able to record complexity in a concise way in the control language is a good thing in most cases, such as where (transformation-)state information is kept in the host graph between applications of the rules. Care must be taken though, that the combination of the rule system and the control language remains understandable. If we move all the complexity to the control expression, we might end up with a perfectly working system which uses rules that cannot be understood properly. The opposite is also true; if we embed all the complexity in the host graph and do not use the control language at all, we are putting the complexity in the wrong place as graph transformation rules should be used to transform the graph, not schedule themselves at the appropriate time.

3.1 Basic input and output parameters

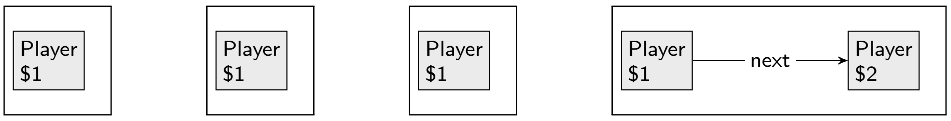
The first extension would be to add basic input ("match this node") and output ("return the matched node") parameters to the control language, allowing us to remember which nodes have been matched between rule applications and controlling which nodes a rule can be applied to. This extension fulfills the first two requirements stated at the start of this chapter; using input parameters will make it easier to specify rules that have to match on certain nodes (rather

```
node current;
get_first_player(out current);
alap {
  throw_die(current);
  move_pawn(current);
  next_player(current, out current);
}
```

Listing 3.1: Control program for Ludo with variables

than placing control "tokens" on the host graph) thus fulfilling the first requirement, while both input and output parameters together will work in fulfilling the second requirement. This extension does not automatically reduce the state space, but it will not make it larger either. Backwards compatibility can be maintained by allowing usage of graph transformation rules without parameters too.

Example 3.1.1 (Ludo with variables) *Consider again the Ludo example presented in section 2.5. Instead of keeping track of the current player in the graph and the rules, we could parameterise this (note that we are not claiming that this is a good thing per se, but it will be an option with this extension). In order to do this, we will need to first capture the node that contains the current player in a variable and subsequently update this variable to reflect each new player in turn. The corresponding control program can be found in listing 3.1 (variable names with* `out` *prepended to them are output variables, without a prefix they are input variables) and the new rules in figure 3.1. The start graph is the same as in the example with control from the aforementioned section, without the current-labelled self-edge on Player 1. Note that this method has the added value of having a random player be the first player. In this very trivial example the first three rules look exactly the same, but bear in mind that in a full implementation each of these rules would do something with the host graph.*



(a) get_first_player (b) throw_die (c) move_pawn (d) next_player

Figure 3.1: The rules in the variables version of Ludo

Deleted nodes If an input variable to a rule tries to match a node that has been removed from the graph by another rule application, one of two things can happen:

1. The rule application can fail, thus blocking the program;
2. the input parameter can be ignored and treated as a "don't care" variable.

We think the former option is the better one as this would provide the most predictable results. The user of the control language will have to keep track of which rules affect which variables and consequently not use variables that point to nodes that have been deleted as input variables.

3.2 Function parameters

As functions in the Groove control language are called in the same way as rules, it would stand to reason that they should also receive parameters. However, care needs to be taken for output parameters, which can be handled in different ways. We recognise five ways to handle output parameters for functions:

1. Ignore them completely.
2. Bind them to the first matching in which they are used, but use it as a free variable afterwards (i.e. return the first match but don't save it within the function).
3. Require that the function uses its output parameters only once.
4. Bind them to the last matching in which they are used.
5. Bind them to the first matching in which they are used and subsequently use these bindings within the function.

Option #1 can clearly not be considered as there are more useful ways to handle output variables. Of the remaining four, we believe option #5 to be the most viable since it most closely resembles the way variables are handled in the previous section. Having as little difference between the way these variables are handled in the various places of the program improves clarity when using the program, which can only be seen as something good.

Input variables can be used in the same way as they are used for single rules, namely that they are bound when the function is called and will be used in every their every instance in the function.

The fulfillment of the extension requirements for this is the same as the basic input/output parameters. It extends on that functionality, making it usable in more places.

3.3 Quantified parameters

As quantified parameters can match zero, one or more nodes, they will need to be treated differently from normal nodes. There is a more distinct difference between input parameters and output parameters in this case.

3.3.1 Output parameters

When a quantified node is used as an output parameter, it might not return just one value. As such, if we want to use quantified nodes as output parameters we will need to make provisions for this. The logical thing to do seems to save all the returned nodes into a list, which requires extra syntax for iterating over lists and possibly other list operations, like unions and intersections of two lists or adding and removing nodes from lists manually.

3.3.2 Input parameters

If a variable is being used as an input-parameter to a quantified node, not much special happens. The node can be regarded as a single node rather than a quantified node. However, when we want to use the output from a quantified parameter as the input for another node, we might end up trying to use a list of zero or more than one value where only one value is expected. In this case, three things can happen:

1. The node it is matched to is a universal quantification node ($\forall$), in which case the rule will be applied while matching the node to each item of the list. If no items exist in the list, this rule will be skipped.
2. The node it is matched to is an existential quantification node ($\exists$), in which case there will be several possible applications for the rule, one for each value in the list (if any). If no items exist in the list, this rule can be skipped.
3. The node it is matched to is a normal node, in which case the same happens as in the second case except at least one match will have to be found else this rule cannot be applied. As such, this rule will block and if there are no other branches the program will end.

This extension can, like the previous two extensions, fulfill all of the requirements set at the start of this chapter. The third requirement once again requires the user to use the extension in a way that fulfills it however.

```
node current;
int counter;
counter = 0;
get_first_player(out current);
alap {
  throw_die(current);
  move_pawn(current);
  counter = counter+1;
  next_player(current, out current, counter, out counter);
}
```

Listing 3.2: Mensch ärgere dich nicht with variables and attributes

3.4 Arithmetic expressions in control language

The next step, after implementing input and output parameters, could be to also incorporate arithmetics into the control language. This would allow us to manipulate attribute nodes directly from the control language, rather than using rules to do so.

Suppose we want to keep track of how many turns have been taken in our Mensch ärgere dich nicht program. Using a counter in the host graph and incrementing this counter in rules would enable us to do this, but it is quite tedious to do so; there will have to be a counter variable kept in the host graph and in each rule that needs to increment this counter you will need at least:

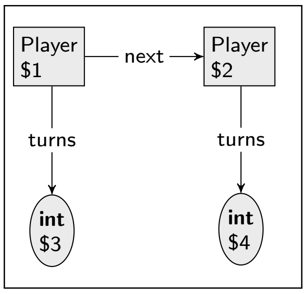
- a static integer value of 1;
- a procedure node;
- a reference to the counter node;
- a reference to the old value of the counter node;
- an attribute node to take new value for the counter node.

It would certainly take a lot less space to put most of this in the control language as an integer variable and just use this variable as an input whenever the value of the counter was needed in the host graph. Care must be taken that we do not move too much complexity from the rules to the control language, such that the rules will become incomprehensible. As this cannot be decided unanimously for all cases, the decision on where to put the complexity would remain with the user of the program.

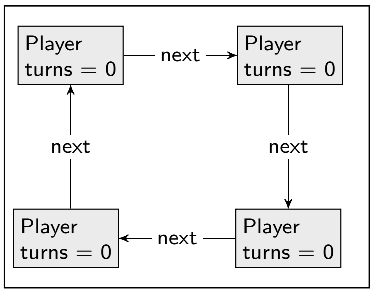
Example 3.4.1 (Ludo with arithmetic expressions) *Consider again the example from section 3.1. Suppose we want to add a counter to every player, keeping track of how many turns they have already played. We could, as described above, implement this by incrementing the counter in the* `next_player` *rule, but this would require a lot of added complexity to this rule.*

Instead here is how we could do it using the control language. We initialize the counter to zero as the first time we increment it, no turns have passed yet. In each of the loops which represents a turn, we first increment the counter, then assign it to the current player and while getting the next player we also get its associated counter variable. The control program for this can be found in listing 3.2 and the new `next_player` *rule as well as the new start graph in figures 3.2a and 3.2b respectively.*

As the requirements from the beginning of this chapter go, this extension can clearly fulfill the first one. It can also fulfill the third one depending on how this feature is implemented, and the fourth one (this extension adds syntax, it does not invalidate the old syntax). The second requirement is a different story however, as this extension moves complexity from the host graph and the graph transformation rules to the control program, but does not necessarily separate the concerns; some complexity might best be left in the rules else it will be unclear what a rule does without looking at the control program. With this extension it is possible to specify complexity in the control program that would most likely be more fitting in the graph and rules.



(a) next_player



(b) start graph

```
boolean entered;
try {
  enter_passenger+;
  entered = true;
} else {
  entered = false;
};

if (entered == true) {
  close_door;
}
```

Listing 3.3: An example of boolean expressions in the control language

3.4.1 Boolean expressions

Boolean expressions are a special case of arithmetic expressions. With these we would be able to set and compare booleans and make entire blocks of the control program conditional. While we can do that now, this would allow us to be even more expressive in this respect.

It could for instance be nice to be able to constrain the execution of a certain rule based on whether another rule had previously been executed. This could be done by adding boolean variables to the language. The boolean literal `true` is already in there, but if we add false and allow conditional statements to use these variables, we could have this feature.

This feature was also described by Michiel Hendriks of the AMPLE project as something he would have liked in his usage of Groove.

An example can be found in listing 3.3. This program tries to apply the rule `enter_passenger` as often as possible (but at least once). If this succeeds it sets the boolean variable `entered` to `true`, if not it sets it to `false`. It then applies the `close_door` rule but only if at least one passenger had entered.

3.5 Choosing the extensions for the project

We have chosen the extensions to be implemented in this project with consideration of how well each of them fulfills the requirements set at the start of this chapter, as well as the time available for the project.

For the first extension we chose basic input/output parameters, which will be detailed in chapter 4. We chose this both because it fulfills all of the requirements and because the other extensions require this extension to be present in order to work.

For the second extension we considered both the function parameters as well as attribute parameters. As we only had time for two extensions in total, and one extension was already spoken for, we could only choose one of these. We decided on attribute parameters as one of the members of the committee, Selim Ciraci, saw a possible application for this in a project he had worked on. Attribute parameters are worked out in chapter 5.

Chapter 4

I/O Parameters

In chapter 2 we have described the control language in Groove as it was before this project started. We will now tackle the extension to this with basic input and output parameters as suggested in section 3.1. For this purpose, we first extend the theory. Both the syntax and the semantics of the control language should be adapted, as well as nearly all the models for the various automata and the graph transformation rules we have.

While we worked out the theory for this chapter, we found that the required adaptations to the models from chapter 2 became more complex than we anticipated and that the implementation of them would cause this project to severely extend its timeframe. To remedy this, we have imposed some restrictions which we will explain near the end of the chapter in section 4.7.

4.1 Modified control language syntax

In order to extend the control language, the first thing that needs to change is the syntax. The various models will then be changed in order to support the new semantics of this syntax. For the extension with I/O parameters, these things have changed in the syntax:

1. Node variables will need to be declared;
2. Rules have a parameter list, with input and output parameters combined.

We have chosen to combine input and output parameters into one list when calling a rule. As both are represented in the same graph when looking at the rule, this is the clearest solution. Output parameters will be prepended by the keyword `out`, while input parameters do not have any prefix. The rest of the syntax follows Java as closely as possible whenever possible. The full new syntax in EBNF notation thus becomes the one shown in listing 4.1, with the bold lines being added. Note that rules can be used either with or without parameters (as per the `call` rule in the syntax), but if parameters are used the number of parameters must be the same as the amount specified by the rule itself. We will call the language produced by this syntax `Lang`.

In this chapter, `Var` will be used to denote the set of possible variables, which as can be seen in listing 4.1 consist of identifiers made up of letters, numbers and a couple of special characters (identifiers have to start with a letter). $\perp$ will be used to indicate dummy variables (e.g. "don't care" variables, or `_` in the control language) wherever applicable. Also, each variable (except $\perp$) may be used as output only once in a single control expression (Cytron et al.[4] explain that any program that has multiple assignments to one variable can be converted into a form where each variable is only assigned to once, called Static Single Assignment form). The reason for this restriction will be explained in section 4.6.

4.1.1 Examples

Some examples of expressions in the extended control language can be found in listings 4.2 and 4.3.

```
program : (function|statement)*
function : 'function' identifier '(' ')' block
block : '{' statement* '}'
condition : conditionliteral ('or' condition)?
statement:
  | 'alap' block
  | 'while' '(' condition ')' 'do'? block
  | 'until' '(' condition ')' 'do'? block
  | 'do' block 'while' '(' condition ')'
  | 'try' block ('else' block)?
  | 'if' '(' condition ')' block ('else' block)?
  | 'choice' block ('or' block)*
  | expression ';'
  | var_declaration ';'
conditionliteral
  : 'true'
  | call
expression : expression2 ('or' expression)
expression2
  : expression_atom ('+' | '*')?
  | '#' expression_atom
expression_atom
  : 'any'
  | 'other'
  | '(' expression ')'
  | call
var_declaration : 'node' identifier (',' identifier)* // item 1
call : identifier ('(' var_list ')')?                // item 2
var_list : variable (',' var_list)?                  // item 2
variable : 'out'? identifier | '_'                   // item 2
identifier
  : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'-'|'_'|'.')*
```

Listing 4.1: The new syntax for the control language

```
node n1, n2, n3;        // declaration of three node variables
a;                      // rule application without parameters
b(out n1);              // rule application with 1 (output) parameter
b(out n2);
c(n1);                  // rule application with 1 (input) parameter
d(n1, n2);              // rule application with two input parameters
e(n1, n2, out n3);      // input and output can also be combined
```

Listing 4.2: An example showing the usage of the new syntax

4.2 Graph transformation rules

In order to accomodate parameters, graph transformation rules require a slight adaptation in their definition. Every rule will have a list of parameters which map to nodes of the left hand side or right hand side. Some of these parameters can be input parameters, some can be output parameters and some can be both. This is determined as follows:

- Parameters in the LHS that also appear in the RHS can be either input or output parameters;
- Parameters in the LHS that do not appear in the RHS (deleted nodes) can only be input parameters;
- Parameters that appear only in the RHS (created nodes) can only be output parameters.

Nodes in the NACs or in nested subrules (which occur when quantified nodes are used) can never be parameters.

Graph transformation rules thus become defined in definition 4.2.1.

```
 1  node n1, n2, n3, n4;
 2  a(out n1);        /* n1 now has a value in all subsequent control
 3                       locations unless the node it points to has
 4                       been deleted */
 5  try {
 6    b(out n2);      // n2 has a value for the remainder of this block
 7  } else {
 8    b(out n3);      // n3 has a value for the remainder of this block
 9  }
10
11  /* n2 and n3 now both have no value anymore because the statements
12     on lines 6 and 8 are not both executed */
13
14  alap {
15    b(out n4);
16  }
17
18  /* while n4 may be assigned to multiple times during execution,
19     this statement is allowed because it is only assigned to
20     once syntactically. */
```

Listing 4.3: An example showing variable scopes

Definition 4.2.1 (graph transformation rule with parameters) *A graph transformation rule P is a tuple $(LHS, RHS, r, NACs, IV, OV)$ in which LHS, RHS, r, $NACs$ are the same as in definition 2.3.1 and:*

- *$IV \in V_{LHS}^*$ is a list of nodes from the LHS which form the input parameters of this rule;*
- *$OV \in V_{RHS}^*$ is a list of nodes from the RHS which form the output parameters of this rule.*

Note that $r \circ IV$ and OV may have some overlap. We defined them as separate lists to more easily generate the product of the system automaton and the control automaton later on.

The matching of rules is the like in definition 2.3.1, but with the addition of input parameters. For each of the input parameters that have a value other than $\perp$ in the control program, the rule must match the node specified by the control program in order to obtain a valid match.

Rule application then proceeds as usual, with any output parameters being bound to the nodes they matched.

When calling a rule from the control program, the rule will only be enabled if the values of the input parameters used by the control program match the values of the input parameters in the rule.

Consider for example the rule presented in figure 4.1, once again with the numbers in red next to the nodes being the node identities for this rule. This rule looks for an node labelled A with a b-edge to a node labelled node B. It then deletes this node labelled B (and the edge pointing to it) and creates a new node labelled C and a c-edge. It only does this, however, if no node with a *Stop* label exists. The two lists of parameters for this rule are as follows: $IV = [2, 4]$ (the nodes with parameters which appear in the left-hand side), $OV = [2, 3]$ (the nodes with parameters which appear in the right-hand side).

4.3 System automaton

In the following definitions, the system automaton defined in section 2.4 will be extended to allow for input and output parameters. To facilitate the creation of the product of the parameterised system automaton and the parameterised control automaton, two lists have been added to each element of Σ:

Definition 4.3.1 (parameterised system automaton) *A parameterised system automaton is a tuple $(Q, \Sigma, \rightarrow, q_0, S)$ where Q, $\rightarrow$, q_0 and S are the same as in definition 2.4.2 but with a transition now being denoted $d_{\mathcal{A}} \xrightarrow{n,i,p,c,I,O}_{\mathcal{A}} d'_{\mathcal{A}}$ where:*

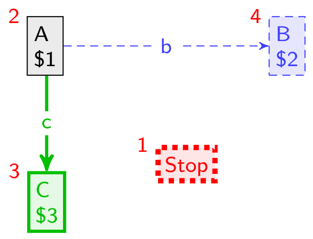
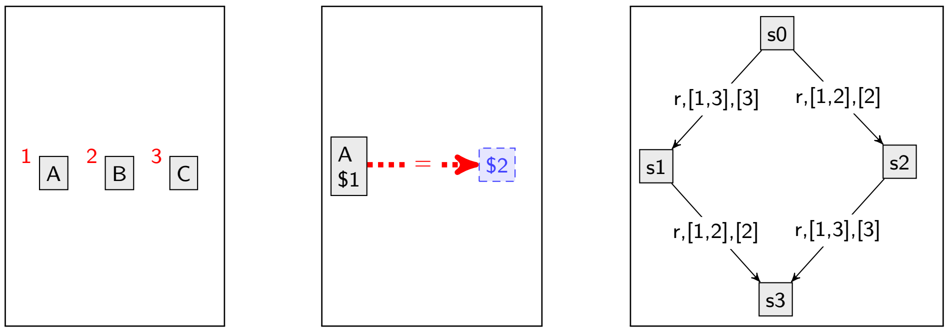


Figure 4.1: An example rule with parameters

- *p is the production morphism, that is the morphism from $d_{\mathcal{A}}$ to $d'_{\mathcal{A}}$;*
- *c is the comatch, that is the morphism from the RHS of the rule to $d'_{\mathcal{A}}$;*
- *$I = i \circ IV_n$ is a list of the matchings for all the input parameters for the rule in a transition;*
- *$O = c \circ OV_n$ is a list of the matchings for all the output parameters for the rule in a transition.*

n and i are unchanged; as in definition 2.4.2 they are the rule name and application identifier respectively.

The production morphism is required to construct the product of this automaton with the parameterised control automaton later on. Note that I and O may have some overlap. An example can be found in figure 4.2. The starting graph is in 4.2a (with the node identities in red numbers next to the nodes), the single rule r in 4.2b and the parameterised system automaton in 4.2c.



(a) starting graph (b) rule r (c) parameterised system automaton

Figure 4.2: An example parameterised system automaton

4.4 Control automaton

In order to allow for parameters, the control automaton also receives a few modifications. First, each state will contain a set of the identifiers of the variables that are bound in that state. Second, each transition will gain two lists of parameters, input and output separated. Furthermore, we will use three new notions for rules with parameters and failures with parameters to shorten our definitions. $\mathsf{DVar} = \mathsf{Var} \cup \{\perp\}$, which indicates the variables combined with the $\perp$ character which we use to denote dummy variables. $\mathsf{PRule} = \mathsf{Rule} \times \{T, F\}^{\mathsf{DVar}^*}$ indicates a rule with input parameters and $\mathsf{PFail} = 2^{\mathsf{PRule}}$ indicate all sets of rules with input parameters to be used as failures.

```
node n1, n2;
r;                 // no parameters used
r(_,_);            // two dont_care parameters
r(out n1, _);      // one output and one dont_care parameter
r(out n2, n1);     // one output and one input parameter
r;                 // once again with no parameters
```

Listing 4.4: Control program corresponding to the automaton in figure 4.3

Definition 4.4.1 (parameterised control automaton) *A parameterised control automaton is a tuple $(Q, \Sigma, \rightarrow, q_0, S)$ with Q, q_0 and S the same as in definition 2.5.1.* $\Sigma = (\mathsf{PRule} \times \mathsf{DVar}^*) \cup \mathsf{PFail}$.

Additionally, transitions will now be denoted $d_{\mathcal{C}} \xrightarrow{n, iVars, oVars}_{\mathcal{C}} d'_{\mathcal{C}}$ in which:

- *$iVars \in (\{T, F\}^{\mathsf{DVar}})^*$: a list of length $|IV_n|$ of expressions over input variables. Each index on the list contains an expression which mentions which variables must match and which must not match on this position. Initially these expressions will only consist of a single variable which must match (or $\bot$), but in the determinisation preprocessing phase more variables will be used. For displaying this in the automaton we will use* `x` *to indicate variable x must match on a certain position and* `!x` *to indicate variable x must not match, combining these subexpressions using $\wedge$;*
- *$oVars \in \mathsf{DVar}^*$: a list of length $|OV_n|$ of output variables. Each index on the list can either contain a dummy variable or the name of a unique variable (no variable may be used more than once in the output of a single rule application).*

As failure transitions will have different inputs for different rules and output is irrelevant in the case of failure, failure transitions will be denoted $d_{\mathcal{C}} \xrightarrow{F}_{\mathcal{C}} d'_{\mathcal{C}}$ in which $F \subseteq \mathsf{PFail}$ is a set of rule names and expressions over input parameters as defined above. The notion of failure will thus be extended to include the condition that a failure will be recorded if a set of rules with the current values of the input parameters cannot be applied, rather than just the set of rules not being applicable.

Definition 4.4.2 (correct parameterised control automaton) *A correct parameterised control automaton $\mathcal{C} = (Q, \Sigma, \rightarrow, q_0, S)$ is a parameterised control automaton for which the output variables of any transition $t \in \rightarrow$ do not contain duplicate variables (except $\bot$, which can occur any number of times).*

From a parameterised control automaton, we can derive a function B which maps all states in the automaton to sets of variables which are bound at those states.

Definition 4.4.3 (bound variables) *Given a parameterised control automaton (or one of its derived automaton types) $\mathcal{C} = (Q, \Sigma, \rightarrow, q_0, S)$, $B_{\mathcal{C}}$ is the smallest available function such that:*

$$B_{\mathcal{C}}(q) = \begin{cases} \{x \in \mathsf{Var} \mid \forall t \in iBu_q : (x \in iVars(t) \vee x \in B(src(t)))\} & if q \neq q_0 \\ \emptyset & otherwise \end{cases}$$

$$iBu_q = \{x \in \rightarrow \mid x = (q', a, I, O, q)\}$$

iBu_q is the input bundle for q, that is the set of transitions that end in q.

An example parameterised control automaton can be found in figure 4.3. The corresponding control program is in listing 4.4. A more involved example, where some states have multiple incoming or outgoing transitions, is in listing 4.5, with the resulting control automaton in figure 4.4. Note that in the final state `n2` is not bound because it was only bound in one of the predecessor states. r is a rule that takes two parameters, the first of which can be either input or output and the second of which can only be input.

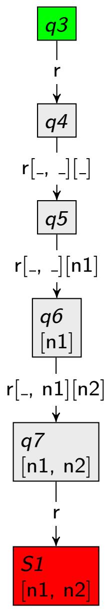


Figure 4.3: An example parameterised control automaton

```
node n1, n2;
r(out n1, _);
choice {
  r(n1, _);
} or {
  r(out n2, _);
  r(n1, _);
}
```

Listing 4.5: Control program corresponding to the automaton in figure 4.4

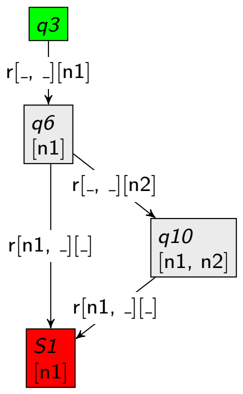


Figure 4.4: Another example parameterised control automaton

4.5 Product automaton

The product of the parameterised system automaton and the parameterised control automaton is constructed in a similar way as before (see section 2.5), while adhering to the added constraints of the input parameters and saving the output parameters. To keep track of the parameter values, every state of the product will have a valuation map $V \in \mathsf{Val}$, where Val is the set of valuations, i.e. the space of all partial functions $\mathsf{Var} \to N$. We use $V_\emptyset$ to indicate the empty valuation function $\emptyset \to N$.

The product automaton is defined by the following rules, an adaptation of definition 2.5.2.

Definition 4.5.1 (parameterised product) *The* product *of a parameterised system automaton $\mathcal{A}$ and a parameterised control automaton $\mathcal{C}$ is defined as $\mathcal{A} \times \mathcal{C} = (Q_\mathcal{A} \times Q_\mathcal{C} \times \mathsf{Val}, \Sigma_\mathcal{A}, \to, (q_{0,\mathcal{A}}, q_{0,\mathcal{C}}, V_\emptyset))$, where the transition relation is defined by the following rules:*

$$\frac{q_\mathcal{A} \xrightarrow{n,i,p,I,O}_\mathcal{A} q'_\mathcal{A} \quad q_\mathcal{C} \xrightarrow{n,iVars,oVars}_\mathcal{C} q'_\mathcal{C} \quad I = v \circ iVars}{(q_\mathcal{A}, q_\mathcal{C}, v) \xrightarrow{n,i} \left(q'_\mathcal{A}, q'_\mathcal{C}, (p \circ v)[O \circ oVars^{-1}]\right)}$$

$$\frac{q_\mathcal{A} \xrightarrow{\lambda}_\mathcal{A} q'_\mathcal{A}}{(q_\mathcal{A}, q_\mathcal{C}, v) \xrightarrow{\lambda} (q'_\mathcal{A}, q_\mathcal{C}, v)}$$

$$\frac{q_\mathcal{C} \xrightarrow{F}_\mathcal{C} q'_\mathcal{C} \quad \forall (n, iVars) \in F : (\neg \exists I \sqsupseteq v \circ iVars : q_\mathcal{A} \xrightarrow{n,i,I,O}_{\mathcal{A}'})}{(q_\mathcal{A}, q_\mathcal{C}, v) \xrightarrow{\lambda} (q_\mathcal{A}, q'_\mathcal{C}, v)}$$

Several notations will require some explanation, namely:

- Any function that can contain $\perp$ is considered to be undefined for those parts of the domain;
- we are allowed to use $(p \circ v)[O \circ oVars^{-1}]$ due to the fact that $oVars$ is injective by definition (see definition 4.4.1) and does not have $\perp$ in its domain;
- we apply p to v in the construction of the new valuation function because nodes might get new identities in case nodes are added or removed. This way the variables still point to the correct nodes. Note that this is only done for the nodes in v that are not overridden by the notation explained in the previous point, as these are taken from O they already have the new identities;
- $v \circ iVars : \mathbb{N} \to N$ becomes undefined for indices where $iVars_i = \perp$.

Originally, states in the product automaton were considered equal if their underlying host graphs were isomorphic and they corresponded to the same state in the control automaton. However, now that variables are introduced, these need to be included in the isomorphism notion.

Definition 4.5.2 (equality on states of a parameterised product) *Two states s and s' of a parametrised product automaton are considered equal if and only if:*

1. *There exists an isomorphism m between the underlying host graphs of s and s';*
2. *s and s' are both considered to be in the same control state;*
3. *$m \circ v_s = v'_s$*

Since the second condition requires that both states correspond to the same control state, both states will also have the same variables so we do not have to check whether all variables exist in both states.

4.6 General determinisation

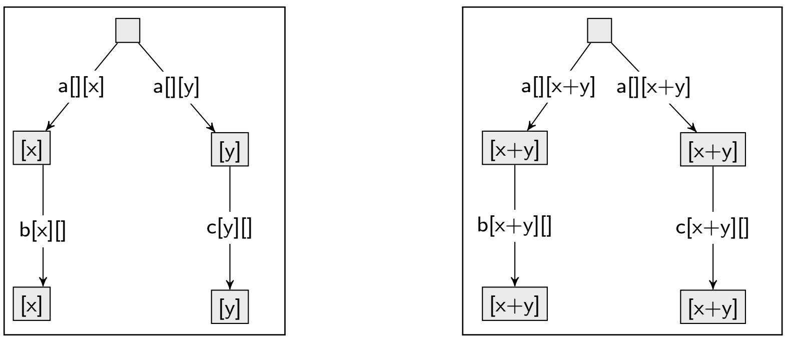
When trying to determinize the parameterised control automaton in the same way as presented in chapter 2.5.1, we are faced with a problem in that a state can have multiple outgoing transitions with the same rule name but different input or output parameters. In the case of different output parameters, this results in a nondeterministic control automaton. In the case of different input parameters, it will also result in a nondeterministic control automaton if the parameters used point to the same nodes. In order to cope with this, we will perform a preprocessing step of determinizing such transitions in order to end up with an automaton that can be determinized using the previously described mechanics. We will discuss output and input parameters separately.

```
choice {
  a(out x);
  b(x);
} or {
  a(out y);
  c(y);
}
```

Listing 4.6: Sample (partial) control expression with output parameters

4.6.1 Preprocessing - output parameters

A special situation occurs when a state has several outgoing transitions with the same rule name but different output parameters. The effect of either transition will be the same except for which parameter will get a value, as in for example figure 4.5a (generated from the expression in listing 4.6). We will call this "determinism through output parameters". In such cases we will replace the parameters with a compound parameter that will be substituted for each of the parameters from then on. The determinisation process will later collapse the two transitions that become identical. We will illustrate this by using an example. In the example we use the notation x+y to denote "$x \vee y$", which is shown in figure 4.5b.



(a) before preprocessing (b) after preprocessing

Figure 4.5: The control automaton corresponding to listing 4.6

In order to further ease this preprocessing step, we also impose a restriction on the control expressions in that they are allowed to syntactically only use a variable as output once. Imposing this restriction enables the assumption that since each variable only occurs as output at most once, we can rename it in the entire automaton without consequence. We do this because renaming a variable in an entire automaton is a much simpler task than deciding in which states we need to rename it. See for instance figure 4.6. In this figure, there exists nondeterminism because the transitions from state 3 to 4 and from state 3 to 5 are the same modulo output variables. We could only rename x and y to $x+y$ in the transitions which introduce the nondeterminism and every transition and state following these, but things will quickly get complicated. Thus we only allow control expressions that generate automata following this definition:

Definition 4.6.1 (unique occurence control automaton) *A unique occurence control automaton is a control automaton $(Q, \Sigma, \rightarrow, q_0, S)$ in which $\forall t, t' \in\rightarrow: \forall v \in oVars_t : v \notin oVars_{t'} \vee t = t'$.*

We also define a variable renaming operation on control automata:

Definition 4.6.2 (control automaton variable renaming) *Given a parameterised control automaton $\mathcal{C} = (Q, \Sigma, \rightarrow, q_0, S)$ and a variable renaming function $f : \mathsf{Var} \rightarrow \mathsf{Var}$, $R(\mathcal{C}, f) =$*

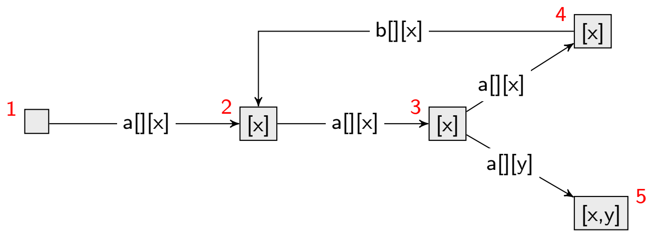


Figure 4.6: Example automaton that requires variable renaming

$(Q, \Sigma, \rightarrow', q_0, S)$ is a renamed control automaton. $\rightarrow' = \{(src_t, n_t, f \circ iVars_t, f \circ oVars_t, tgt_t) \mid t \in \rightarrow\}$. Note that f must be defined for all $v \in \{iVars_t \mid t \in \rightarrow\}$.

Now we can begin our renaming process.

Definition 4.6.3 (output parameter preprocessing) *Given a unique occurence control automaton $\mathcal{C} = (Q, \Sigma, \rightarrow, q_0, S)$ and a variable renaming function f such that in $odet(\mathcal{C}) = R(\mathcal{C}, f) = (Q, \Sigma, \rightarrow', q_0, S)$ the following holds:*
$\forall q \in Q, a \in \mathsf{Rule} : \forall b, b' \in Bu_{q,a} : b = b' \vee oVars_b = oVars_{b'}$
(where $Bu_{q,a} = \{x \in \rightarrow \mid q \Longrightarrow q' \xrightarrow{a,I,O} q''\}$). $odet(\mathcal{C}) = R(\mathcal{C}, f)$ then is an automaton in which no nondeterminism through output variables exists.

Proposition 4.6.4 *A function f as described in definition 4.6.3 exists for all unique occurence control automata.*

Proof 4.6.5 *The function can be obtained from an automaton by, whenever a state has multiple outgoing transitions with the same rule name and input variables, unifying the variables for each spot in the output variable list. Thus, if for instance if a state has several of these transitions with output parameter lists $\{[a, b, \perp], [\perp, c, d], [\perp, \perp, e]\}$, $\{a \mapsto a, b \mapsto x, c \mapsto x, d \mapsto y, e \mapsto y\}$ will be in f (x and y were randomly picked, these should be variable names that were not in the original automaton).* ■

Proposition 4.6.6 (renaming correctness) *Given a unique occurence control automaton $\mathcal{C}$, $odet(\mathcal{C})$ is a correct parameterised control automaton as in definition 4.4.2.*

Proof 4.6.7 *Because in a unique occurence control automaton (see definition 4.6.1) every variable may only appear as output on one transition, and given the fact that the renaming function used in the output parameter preprocessing only renames several variables to the same name if they are used in the same bundle and on the same position in the output parameter list, it will never happen that the list of output variables of a transition contains duplicates (except for $\perp$).*■

4.6.2 Preprocessing - input parameters

As with output parameters, another special situation occurs when a state has several outgoing transitions with the same rule but different input parameters.

As an example, suppose we have an arbitrary rule a which takes one parameter. We also have three rules named b, c and d which do not have parameters (or no parameters are used with them in this example). Consider the control automaton in figure 4.7 (generated by the expression in listing 4.7). If while constructing the product x_1 and x_2 have different values there is no problem, but if they have the same value we will have introduced nondeterminism because the transitions will now be identical (we will call this “nondeterminism through input parameters”). As we do not know which of these two cases will occur at the time of the construction of the control automaton, we cannot simply rename the variables as we did with output parameters. We will instead solve this by taking the subsets of all variables involved and creating transitions for those

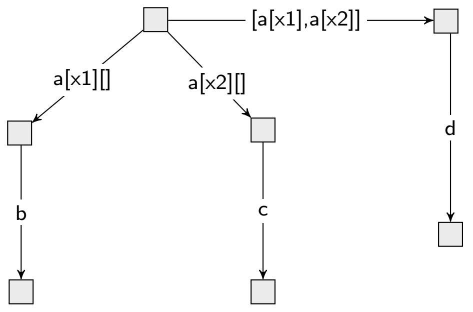


Figure 4.7: An example parameterised control automaton

```
try {
  choice {
    a(x1); b;
  } or {
    a(x2); c;
  }
} else {
  d;
}
```

Listing 4.7: Sample (partial) control expression with input parameters

instead of the ones currently in the automaton. The new transitions will lead to sets of states from the old automaton. In this section, we use the notation $a[x_1 \wedge !x_2]$ to indicate that rule a can be applied with parameter x_1 but not with parameter x_2.

Formally this preprocessing is defined as follows.

Definition 4.6.8 (input parameter preprocessing) *Given a parameterised control automaton $\mathcal{C} = (Q, \Sigma, \rightarrow, q_0, S)$, $idet(\mathcal{C}) = (Q', \Sigma', \rightarrow', q_0, S')$ is another parameterised control automaton, with:*

$$\begin{aligned}
Q' &= \{tgt(t) \mid \forall q \in Q, a \in \Sigma : t \in T_{q,a}\} \cup \{q_0\} \\
\Sigma' &= (\mathsf{PRule} \times \mathsf{DVar}^*) \cup \mathsf{PFail} \\
\rightarrow' &= \{(P, a, I, O, P') \mid P \in Q' \wedge \exists (q, a, I, O, P') \in T_{q,a} : q \in P\} \\
T_{q,a} &= \left\{ (q, a, iV, O, targets_{iV,q,a}) \,\middle|\, iV \in \{f_i \in P_{i,q,a}\}_{1 \le i \le |iV(a)|} \wedge targets_{iV,q,a} \neq \emptyset \right\} \\
targets_{iV,q,a} &= \{tgt(b) \mid b \in Bu_{q,a} \wedge iVars(b) \subseteq iV\} \\
S' &= \{s \in Q' \mid \exists s' \in s : s' \in S\} \\
P_{i,q,a} &= \{T, F\}^{X_{i,q,a}} \\
X_{i,q,a} &= \{iVars_i(b) \mid b \in Bu_{q,a}\} \setminus \{\bot\} \\
Bu_{q,a} &= \{x \in \rightarrow \mid q \Longrightarrow q' \xrightarrow{a,I,O} q''\} \\
A \subseteq B &\Leftrightarrow \forall i \in \{x \in \mathbb{R} \mid 0 \le x \le |A|\} : \forall a \in A_i : B_i(a) = T
\end{aligned}$$

Proposition 4.6.9 *If $\mathcal{C}$ is a parameterised control automaton, $odet(\mathcal{C})$ is a parameterised control automaton in which no nondeterminism through input variables exists.*

The preprocessed version of the automaton from figure 4.7 is in figure 4.8. Note that there is no outgoing transition with the label $a[!x1 \wedge !x2]$ because there is no transition in the original automaton to match this.

This preprocessing step will obviously explode the state space of the control automaton rather quickly when more than two variables are in use or when there are multiple parameters per rule-

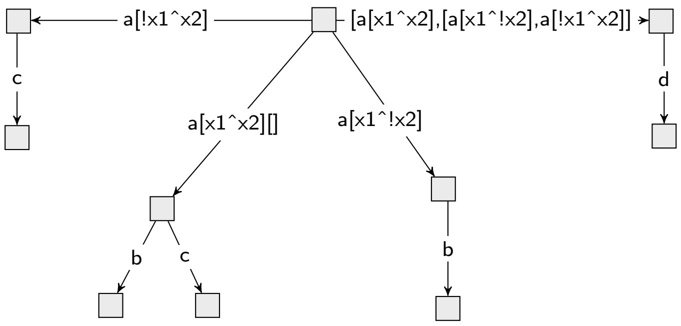


Figure 4.8: The automaton from figure 4.7 after the pre-pass

call, but it should still provide less complexity than other determinisation methods. In most cases many of the combinations of variables will not actually be applicable so they will not appear in the product.

After this preprocessing phase we end up with a preprocessed control automaton with parameters. In this automaton, a state will never have two outgoing transitions with the same rule name and input parameters or the same rule name and output parameters.

Definition 4.6.10 (quasi-deterministic control automaton with parameters) *A quasi-deterministic control automaton with parameters $\mathcal{C}$ is a control automaton with parameters which satisfies the following property:* $q \xrightarrow{n,I,O} q' \wedge q \xrightarrow{n,I,O} q'' \implies q' = q''$

Proposition 4.6.11 *Given a control automaton with parameters $\mathcal{C}$, $idet(odet(\mathcal{C}))$ is a quasi-deterministic control automaton with parameters.*

4.6.3 Determinisation

After the preprocessing phase determinisation can commence. Every transition will still have a failure set and an applicability set, however these sets are now not only sets of rule names but sets of rule names combined with input parameters. Thus each rule name might appear more than once in either set, with distinct input parameters. The application of the rule will also affect the value of certain variables, so output parameters will be added to the transitions as well. Each transition will thus be denoted $q \xrightarrow{[F|A]n,I,O} q'$ where F and A are sets of pairs (r, I) in which furthermore r is a rule name and I is a list of input parameters corresponding with the rule. n is the rule name to be applied in the transition and O is a list of variables corresponding to the output parameters of the rule (note that some positions in both the I and O lists can be empty to indicate "don't care" parameter positions).

Definition 4.6.12 (guarded control automaton with parameters) *A guarded control automaton with parameters is an automaton with* $\Sigma = \mathsf{PFail} \times \mathsf{PFail} \times (\mathsf{PRule} \times \mathsf{DVar}^*)$ *and* $S \subseteq Q \times \mathsf{PFail}$. *Transitions should satisfy the following constraints for all $q \in Q$:*

1. $q \xrightarrow{[F|A]n,I,O}$ *implies* $F \cap A = \emptyset$
2. $q \xrightarrow{[F_1|A_1]n,I,O}$ *and* $q \xrightarrow{[F_2|A_2]n}$ *implies* $F_1 \cup A_1 = F_2 \cup A_2$
3. $q \xrightarrow{[F|A]n,I,O}$ *implies* $\exists q \xrightarrow{[F \cup A]n,I,O}$

Given the new notion for failure as described in definition 4.4.1, the definition for the failure dependency set does not change (see definition 2.5.4), though it will take the new notion of failure into account. Consequently, the definition for control automaton determinisation does not change either, it merely uses the new failure dependency.

Definition 4.6.13 (parameterised control automaton determinisation) *Given a preprocessed parameterised control automaton $\mathcal{C}$, $det(\mathcal{C})$ is a deterministic control automaton with $Q = 2^{Q_\mathcal{C}} \setminus \emptyset$, $q_0 = \{q_{0,\mathcal{C}}\}$, $S = \{(q, \cup_i F_i) \mid \exists q_\mathcal{C} \in q : q_\mathcal{C} \xrightarrow{F_1 \ldots F_n} q'_\mathcal{C} \in S_\mathcal{C}\}$, and $\rightarrow$ is defined by:*

$$\frac{F \subseteq fd(q, n) \quad A = fd(q, n) \setminus F}{q \xrightarrow{[A|F]n} \{q'_\mathcal{C} \mid q_\mathcal{C} \in q \xrightarrow{F_1 \ldots F_n} \xrightarrow{n} q'_\mathcal{C}, F_1 \cup \ldots \cup F_n \subseteq F\}}$$

Constructing the product of the guarded parameterised control automaton and the parameterised system automaton will now need to apply the new failure rules. Rather than checking if a set of enabled rule-names is contained within the set of all enabled rule-names, it is now required that the every rule in A is applicable *using the specified input parameters*. The definition for the guarded product (definition 2.5.7) can remain the same if we adapt the definition for enabled rules:

Definition 4.6.14 (enabled rules with parameters) *Given a parameterised system automaton $\mathcal{A}$, the function enabled : $Q_\mathcal{A} \rightarrow \mathsf{Rule} \times 2^{\mathsf{Node}}$ is defined as:*

$$enabled(q_\mathcal{A}) = \{(n, in) \in \mathsf{Rule} \times 2^{\mathsf{Node}} \mid \exists i \in \mathsf{Id} : q_\mathcal{A} \xRightarrow{(n,i,in)}_\mathcal{A}\}$$

The guarded product definition can then be used as before.

4.7 Restrictions imposed upon control expressions

During the course of this research we found that the scope kept expanding due to the fact that we wanted to allow all possible control expressions to be used. This is illustrated in the following sections. In order to reduce the scope, we decided to impose a restriction on the expressions we allow. In short, we reject all expressions that specify nondeterminism when using parameters.

We propose that this does not adversely affect the working of Groove as this type of nondeterminism is seldom needed. We back this empirically by noting that all of the inspected related tools (see section 8.2) do not allow nondeterminism in their control structures at all. Groove does currently allow this, but for now we will disallow the specification of nondeterminism in the control language when parameters are used.

Some expressions that contain nondeterminism can easily be rewritten in a deterministic way. For example, the expression `choice{a; b;}or{a; c;}` can be rewritten as `a; choice{b;}or{c;}`. For other expressions, such as the one presented in figure 2.16 (`choice {a; c;}or{try{b; c;}else{a;}}`), a deterministic version can not be constructed. We can approximate the same behaviour but the language described will always be different. This is because to properly determinize this expression we would need to be able to specify rules that must be applicable in order to take a certain transition - as is described in section 2.5.

So, even though not all nondeterministic expressions can be written deterministically, we restrict Groove control expressions with parameters to deterministic ones. Formally this restriction is specified in definition 4.7.1.

Definition 4.7.1 (accepted parameterised control automata) *An accepted parameterised control automaton is a $(Q, \Sigma, \rightarrow, q_0, S)$ such that:*
$\forall q, q', q'', v, v' \in Q, r \in \mathsf{Rule} : (q \Rightarrow q' \xrightarrow{r} v, q \Rightarrow q'' \xrightarrow{r} v') \implies q' = q'' \wedge v = v'$, *where $a \Rightarrow a'$ denotes that a path exists from a to a' consisting of zero or more failure transitions. Note that this includes $q = q'$ and $q = q''$.*

We define whether each control statement can terminate instantly in definition 4.7.3. This is required to correctly define the list of initial actions for a given control statement in definition 4.7.5, which we then use to determine whether an expression will generate an accepted parameterised control automaton.

As conditions can be either rules or *true*, we first define a predicate that will tell us whether a condition is the former or the latter, to be used in the definition of instantly terminating control expressions.

Definition 4.7.2 (condition is a rule) *Given a condition C of a control expression, isRule(C) defines whether this condition is the expression* true *or a rule-application (or a composite of rule applications). isRule(C) holds if and only if $C \neq$ "true".*

Definition 4.7.3 (instantly terminating control expressions) *The predicate $IT(\mathcal{E})$ defines whether an expression $\mathcal{E}$ can terminate instantly, meaning it is possible that it will not apply a graph transformation rule.*

$$\begin{array}{rcl}
\mathit{IT(alap\ A)} & = & \mathit{true} \\
\mathit{IT(while\ C\ do\ A)} & = & \left\{\begin{array}{ll} IT(C) & \mathit{if\ isRule}(C) \\ \mathit{false} & \mathit{if}\ \neg \mathit{isRule}(C) \end{array}\right. \\
\mathit{IT(until\ C\ do\ A)} & = & \left\{\begin{array}{ll} IT(C) & \mathit{if\ isRule}(C) \\ \mathit{true} & \mathit{if}\ \neg \mathit{isRule}(C) \end{array}\right. \\
\mathit{IT(do\ A\ while\ C)} & = & \left\{\begin{array}{ll} IT(A) \wedge IT(B) & \mathit{if\ isRule}(C) \\ \mathit{false} & \mathit{if}\ \neg \mathit{isRule}(C) \end{array}\right. \\
\mathit{IT(try\ A)} & = & \mathit{true} \\
\mathit{IT(try\ A\ else\ B)} & = & \mathit{IT(B)} \\
\mathit{IT(if\ C\ then\ A)} & = & \left\{\begin{array}{ll} IT(C) & \mathit{if\ isRule}(C) \\ IT(A) & \mathit{if}\ \neg \mathit{isRule}(C) \end{array}\right. \\
\mathit{IT(if\ C\ then\ A\ else\ B)} & = & \left\{\begin{array}{ll} IT(B) & \mathit{if\ isRule}(C) \\ IT(A) & \mathit{if}\ \neg \mathit{isRule}(C) \end{array}\right. \\
\mathit{IT(choice\ A\ or\ B)} & = & \mathit{IT(A)} \vee \mathit{IT(B)} \\
\mathit{IT(var\_declaration)} & = & \mathit{true} \\
\mathit{IT(rule)} & = & \mathit{false} \\
\mathit{IT(rule+)} & = & \mathit{IT(rule)} \\
\mathit{IT(rule^*)} & = & \mathit{true} \\
\mathit{IT(\#rule)} & = & \mathit{true} \\
\mathit{IT(rule \mid rule2)} & = & \mathit{IT(rule)} \vee \mathit{IT(rule2)} \\
\mathit{IT(A;\ B)} & = & \mathit{IT(A)} \wedge \mathit{IT(B)} \\
\mathit{IT(nil)} & = & \mathit{true} \\
\mathit{IT(true)} & = & \mathit{true} \\
\mathit{IT(any)} & = & \mathit{false} \\
\mathit{IT(other)} & = & \mathit{false}
\end{array}$$

Proposition 4.7.4 *Given a control expression $\mathcal{E}$, if $[\![\mathcal{E}]\!] = (Q, \Sigma, \rightarrow, q_0, S)$, then $IT(\mathcal{E})$ if and only if $\exists q \in S : q_0 \Rightarrow q$.*

Using definition 4.7.3, we can now define the multiset of initial actions for a control expression.

Definition 4.7.5 (initial actions) *The initial actions for a control statement $\mathcal{E}$, init$(\mathcal{E})$, are a multiset of rule names which can occur as the first action of the statement, after any non-rule-application actions have occurred. We split this up as follows:*

$$
\begin{array}{rcl}
\textit{init(alap A)} & = & \textit{init(A)} \\
\textit{init(while C do A)} & = & \begin{cases} init(A) & if\ \neg isRule(C) \\ init(C) & if\ isRule(C) \wedge \neg IT(C) \\ init(C) + init(A) & if\ isRule(C) \wedge IT(C) \end{cases} \\
\textit{init(until C do A)} & = & \begin{cases} \emptyset & if\ \neg isRule(C) \\ init(C) & if\ isRule(C) \end{cases} \\
\textit{init(do A while C)} & = & \begin{cases} init(A) & if\ \neg isRule(C) \\ init(A) & if\ isRule(C) \wedge \neg IT(A) \\ init(C) + init(A) & if\ isRule(C) \wedge IT(A) \end{cases} \\
\textit{init(try A)} & = & \textit{init(A)} \\
\textit{init(try A else B)} & = & \textit{init(A) + init(B)} \\
\textit{init(if C then A)} & = & \begin{cases} init(A) & if\ \neg isRule(C) \\ init(C) & if\ isRule(C) \wedge \neg IT(C) \\ init(C) + init(A) & if\ isRule(C) \wedge IT(C) \end{cases} \\
\textit{init(if C then A else B)} & = & \begin{cases} init(A) & if\ \neg isRule(C) \\ init(A) + init(C) & if\ isRule(C) \wedge \neg IT(C) \\ init(B) + init(C) & if\ isRule(C) \wedge IT(C) \end{cases} \\
\textit{init(choice A or B)} & = & \textit{init(A) + init(B)} \\
\textit{init(var\_declaration)} & = & \emptyset \\
\textit{init(rule)} & = & [\ \textit{rule}\ ] \\
\textit{init(rule+)} & = & [\ \textit{rule}\ ] \\
\textit{init(rule*)} & = & [\ \textit{rule}\ ] \\
\textit{init(\#rule)} & = & [\ \textit{rule}\ ] \\
\textit{init(rule or rule2)} & = & [\ \textit{rule, rule2}\ ] \\
\textit{init(A; B)} & = & \begin{cases} init(A) & if\ \neg IT(A) \\ init(B) + init(A) & if\ IT(A) \end{cases} \\
\textit{init(true)} & = & \emptyset \\
\textit{init(nil)} & = & \emptyset \\
\textit{init(any)} & = & \mathsf{Rule} \\
\textit{init(other)} & = & \emptyset
\end{array}
$$

Proposition 4.7.6 *Given a control expression $\mathcal{E}$, if $[\![\mathcal{E}]\!] = (Q, \Sigma, \rightarrow, q_0, S)$ then $\exists q \in Q : q_0 \Rightarrow q \xrightarrow{a}$ if $a \in init(\mathcal{E})$.*

Following from this we define acceptable control expressions as follows:

Definition 4.7.7 (accepted control expressions) *An accepted control expression $\mathcal{E}$ is one in which for every $s \in statements_{\mathcal{E}}$, $init(s)$ contains no duplicate rule names.*

Proposition 4.7.8 *Given a control expression $\mathcal{E}$, if $\mathcal{E}$is accepted in the sense of definition 4.7.7, $[\![\mathcal{E}]\!]$ will always be accepted in the sense of the definition 4.7.1 and can be used by Groove without extra determinisiation.*

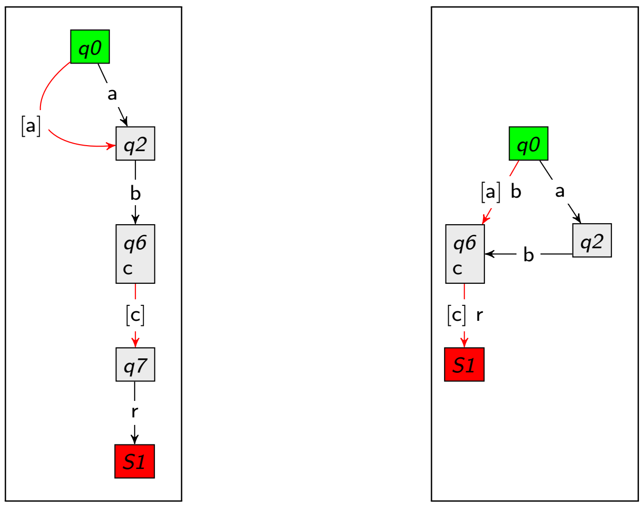
Proof 4.7.9 *This can be verified by checking that for each of the statements listed in definition 4.7.5, if the initial actions have no duplicates the resulting automaton for this statement can never contain nondeterminism. Given propositions 4.7.4 and 4.7.6, and as all possible statements are listed, we can prove using induction that if an expression $\mathcal{E}$satisfies definition 4.7.7, $[\![\mathcal{E}]\!]$ will also satisfy definition 4.7.1.* ∎

Example expressions that become invalid with this constraint are for instance `try{a;} else {a;}` and `choice{a}or{choice{b;} or {a;}}` and so on. We do this to prevent having to handle the exceptions that have been mentioned earlier in this chapter. While solutions for these occurrences of nondeterminism have been detailed, we will start by implementing the more restricted case.

4.8 Collapsing failure transitions

As we now only allow control expressions which result in a deterministic control automaton, we have no further need for failure transitions. We can instead collapse these with the rule-applying transitions that follow them, resulting in an automaton that for every transition applies exactly one rule.

Example 4.8.1 *We examine control expression* $\mathcal{E} =$ `try { a; } b; #(c); r;`. $[\![\mathcal{E}]\!]$ *is shown in figure 4.9a. After performing this collapse step, we end up with the automaton in figure 4.9b. The path* $q0 \xrightarrow{[a]} q2 \xrightarrow{b} q6$ *has been replaced by one transition* $q0 \xrightarrow{[a]b} q6$*. Similarly,* $q6 \xrightarrow{[c]} q7 \xrightarrow{r} S1$ *has been replaced by* $q6 \xrightarrow{[c]r} S1$*. Because* $q7$ *is now unreachable, this state has been removed along with its outgoing transition to* $S1$*.*



(a) before rewriting (b) after rewriting

Figure 4.9: Example showing control automaton rewriting

While in the example the automaton reduced in size by this change, it will in most cases make the automaton blow up exponentially, because every failure transition will be replaced by any number of new transitions. We accept this because it makes the execution of the automaton easier. In the implementation before this project started, the exploration could be in one of several control states at once, known as a control location (see section 2.5.2). We had to check whether we were in one of the states of this set and then find out where a transition would lead, also resulting in a set of states. We reduce the amount of operations done by ensuring we will always be in one uniquely identifiable control state instead of a set of control states.

Formally, this transformation is implemented as in definition 4.8.1

Definition 4.8.1 (failure-collapsed control automaton with parameters) *Given a parameterised control automaton (see definition 4.7.1)* $\mathcal{C} = (Q, \Sigma, \rightarrow, q_0, S)$*,* $\mathcal{C}' = coll(\mathcal{C}) = (Q', \Sigma', \rightarrow', q_0, S')$ *is a control automaton, such that:*

- $Q' = \{tgt_t \mid t \in \rightarrow'\} \cup q_0$
- $\Sigma' = \mathsf{PFail} \times (\mathsf{PRule} \times \mathsf{DVar}^*)$
- $\rightarrow' = \{q \xrightarrow{[fail(p)]n,I,O} q' \mid \forall p \in paths_{\mathcal{C}} : trans(p) = q \xrightarrow{n,I,O} q'\}$
- $S' \in Q' \times 2^{\mathsf{Rule}} = \{q \in Q', F \mid q \overset{F}{\Longrightarrow} q' \wedge q' \in S\}$
- $paths_{\mathcal{C}} = \{q \overset{F}{\Longrightarrow} q' \xrightarrow{n,I,O}_{\mathcal{C}} q'' \mid q \xrightarrow{F_1}\xrightarrow{F_2} \dots \xrightarrow{F_k} q', \quad F = \bigcup_i F_i\}$

- *trans*$(p) = q' \xrightarrow{n,I,O} q''$, *where* $p = (q \overset{F}{\Longrightarrow} q' \xrightarrow{n,I,O} q'') \in paths_{\mathcal{C}}$
- *fail*$(p) = \{F \mid p = q \overset{F}{\Longrightarrow} q' \xrightarrow{n,I,O} q''\}$

$paths_c$ is the set of possible paths $q \overset{F}{\Longrightarrow} q' \xrightarrow{n,I,O} q''$ where $q \overset{F}{\Longrightarrow} q'$ indicates a path with only failure transitions from q to q', failing on the rules in F in some order. trans(p) is then the transition part of a path p (i.e. the tail of the path) and fail(p) are the failures of a path p, i.e. the failures on all transitions except the tail of p.

We then call $\mathcal{C}'$ a failure-collapsed control automaton with parameters.

Proposition 4.8.2 *Given an accepted parameterised control automaton $\mathcal{C}$, coll$(\mathcal{C})$ is a control automaton where for every transition $(q \xrightarrow{[F]n,i,O} q') \in \rightarrow'$ exactly one rule n is applied, and each transition is guarded by zero or more rules F, all of which may not be applicable in order to take the transition. Additionally, final states $(q, F) \in S$ are now also guarded by zero or more failures F, indicating that q is only considered a final state if all of the rules in F are not applicable.*

Chapter 5

Attribute Parameters

As an extension to the basic input and output parameters described in the previous chapter, we now extend the theory to allow the usage of attribute nodes as variables. This allows us to input attributes directly from the control program, thus improving the expressiveness even further. We also allow attribute values to be used as input and output parameters just like ordinary nodes were used in the previous chapter. Formerly, attributes could only be inserted by editing the rules or changing the host graph, while it is desirable to be able to use arbitrary values for attribute nodes directly from the control language.

5.1 Problem description

Rules in Groove can perform operations on integers, real numbers, booleans and strings. These are done by specifying a product node, one or more attribute nodes as arguments and a target attribute node. While usually the values of all these nodes are static, it may sometimes be desirable to be able to input their values directly, rather than having to go into the rule editor to change the rules.

Let us consider a game of Pac-Man[19], in which the player has to eat pills while avoiding ghosts that hunt her. The pills are worth points which are added to the score every time a pill is eaten. We could imagine a graph transformation rule that allows the player to eat a pill in an adjacent square, adding 100 points, like in figure 5.1. (Note that the edge that connects the two squares has the label `nextTo|-nextTo`. This is a regular-expression label which indicates that this edge should match an edge in the host graph with the `nextTo` label, or an edge in the opposite direction with the same label.)

If we also have other rules which modify the score of a player when similar events occur (for instance, when the player eats fruit instead of a pill, or runs over a blue ghost), we will have the scoring system divided over several rules. While this is not a problem per se, if we wanted to

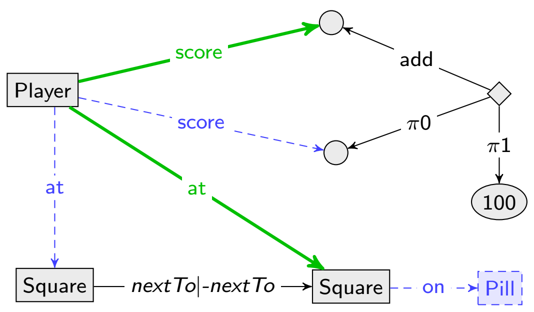


Figure 5.1: The eat_pill rule, which makes the player eat a pill on an adjacent square and adds 100 points to the score

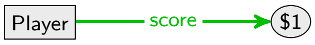


Figure 5.2: The set_score rule could be used to initialize the player's score

for instance double the worth of every eaten object, we would have to edit each of these rules to reflect the new value.

We can solve the problem by parameterizing attributes similarly to how we parameterized nodes. That would still not allow us to input the values for each of the eaten objects in Pac-Man from the control program though, since we would have to get them from the graph into a variable first.

However, while we can not "create" nodes in the control program since we do not know their identity, it is possible to directly write the values of attributes since they are derived from primitives and their identity does not change. Thus we will also allow attribute parameters to be input the same way as primitives in Java. We could then for instance write `eat_pill(200);` to have the player eat a pill and add 200 to her score. Several problems arise with this approach that have to be dealt with:

1. Rules can currently specify that they expect an attribute somewhere, without stating what type of attribute it should be. This could pose problems, since while the rule itself might not fail due to an attribute of the wrong type being passed, subsequent rules might expect a value of a specific type to be there. For instance, if we were to apply the `set_score` rule from figure 5.2 and passing it a string parameter, there would be no problem. If we then try to apply the `eat_pill` rule again, it would not match because it can not add an integer to a string.
2. Groove does not currently allow arbitrary nodes to be parameterized. Particularly, if we have an isolated attribute node (an attribute node that has no matching edges, only creator edges), that node must have a concrete value - not a parameter. The way Groove currently implements attribute nodes is that they are virtually always present; this means that if we try to match an attribute node without specifying a value, it could potentially match anything. However, we do want to be able to use attribute nodes without a concrete value, which will get a value when the rule is matched in combination with a control program.

To address the first problem, we could either ignore the problem (and have programs end in a semi-unexpected manner) or not allow attribute parameters to match arbitrary attribute types. Instead, when writing a rule with attribute parameters, the user should specify the type of parameter that is expected. We choose to use the latter approach in having the user explicitly define the desired behaviour.

The solution to the second problem has more to it however. If we want to allow isolated attributes to be used as parameters, we will have to ensure they always have a value when the rule is applied. This has two implications:

1. The parameter can never be used as output;
2. The parameter *must* be present when this rule is used. As such, whenever the rule is used in a control program, this parameter must be present. Additionally, this means that the rule will be invalid when no control is used (without control, all rules are scheduled at all times).

We can solve both these implications, while maintaining backward compatibility. Users who do not use attribute parameters will not notice the difference, but users who do want to use them can do so.

5.2 Solutions

To solve the problems stated in the previous section, we will make modifications to both the rules and the control language. Combined, these will allow both direct input of attributes from the control language and getting attributes as variables and re-using them later.

5.2.1 Rule modifications

In order to solve the problem of not being able to use isolated attribute nodes as parameters, we ask that the user specifies these nodes as required-input nodes. To this end, we will include additional keywords for specifying parameters to be restricted to input or output, namely `parin:` for input-only parameters and `parout:` for output-only parameters (`par:` is used to indicate parameters that may be used as both input and output). If `parin:` is used on a parameter that can only be output (i.e. a node that only occurs in the RHS), an error will be raised. The same goes for using `parout:` on nodes that are only in the LHS. An error will also be raised if a control program attempts to use a parameter in a way that was specified to not be available in the rule.

Additionally, using `parin:` will make that parameter required. This means that Groove will consider the rule system invalid if it has (active) rules which require input parameters but no control is present. It also means that if control is present and it calls rules that have required input parameters, there will be an error if these input parameters are not provided. These effects allow us to remove the condition that there are no isolated attributes in rules, replacing it with the condition that there are no isolated attributes *that are not required input parameters*, thus ensuring correct functionality.

To ensure that a control program can only provide an attribute of a proper type, we require that users who want to make attribute parameters type these, using the `bool:`, `int:`, `real:` or `string:` keywords rather than `attr:` which is used for general (unparameterized) attributes. While this adds some extra effort and explicitness, we conjecture that attribute nodes are rarely if ever used for multiple types so it will not affect the expressiveness of Groove.

5.2.2 Control language modifications

While the control syntax itself does not change much to incorporate attribute parameters, its semantics do. Type checking has been added for attribute parameters and the control parser now checks for required inputs. The new features in the control language are as follows:

- Variables can now not only be declared as `node`, but also as `bool`, `int`, `real` and `string`, matching their relevant typed attribute nodes;
- These new variable types can be used as input or output parameters in rules that allow this. As such, we can obtain the values of attribute nodes from the host graph by using an output parameter and subsequently re-use it as an input parameter;
- We can use literal attributes as input parameters much like in programming languages such as Java. Rules that expect a `bool` variable will accept verb—true— or `false`, rules that expect an `int` will take values such as `1337` or `-187`, rules that expect a `real` will accept `3.1415`, `.5`, `10.` and so on, and rules which expect a `string` will accept anything that has been put between double quotes and does not contain unescaped double quotes or backslashes. Examples of this are `"Hello, world!"` and `"I say: \\n\"Good day!\""`;
- The syntax checker will verify that types are used correctly by checking the types that rules expect for every parameter, as well as checking that input parameters are not used for output-only nodes and vice versa;
- The syntax checker will also verify that if a rule is used that has required input parameters, the rule should is not used without parameters and the required parameters are present.

The changes in the syntax can be found in figure 5.3 (where `~X` means "anything but `X`"). Additionally it should be noted that a real number may not consist of just a period; there should be digits before it or after (or both). The new control automaton is in definition 5.2.1.

Definition 5.2.1 (parameterised control automaton with attributes) *A parameterised control automaton with attributes is a tuple* $(Q, \Sigma, \to, q_0, S)$ *with* Q, $\to$, q_0 *and* S *the same as in definition 4.4.1.* $\Sigma = \Big(\mathsf{Rule} \times (\{T, F\}^{(\mathsf{DVar} \cup \mathsf{Attr})})^* \times \mathsf{DVar}^*\Big) \cup \Big(\mathtt{Fail} \times ((\{T, F\}^{(\mathsf{DVar} \cup \mathsf{Attr})})^*\Big)$.

```
var_declaration : var_type identifier (',' identifier)*
var_type        : 'node' | 'bool' | 'int' | 'real' | 'string'
variable        : 'out'? identifier | '_' | literal
literal
  : bool_literal | int_literal | real_literal | string_literal
bool_literal    : 'true' | 'false'
int_literal     : number+
real_literal    : number* '.' number*
string_literal  : quote string_char* quote
quote           : '"'
string_char     : ~(quote | backslash)
                | backslash (quote | backslash)
number          : '0' .. '9'
backslash       : '\'
```

Figure 5.3: The (partial) syntax for the control language extended with attribute parameters

```
setCounter(5);
while(reduceCounter(1)) do {
  // some loop here
}
```

Listing 5.1: Example control program with literal attribute parameters

Additionally, transitions will now be denoted $d_{\mathcal{C}} \xrightarrow{n,iVars,oVars}_{\mathcal{C}} d'_{\mathcal{C}}$ *in which:*

- $iVars \in (\{T,F\}^{(\mathsf{DVar} \cup \mathsf{Attr})})^*$ *is the same as in definition 4.4.1, with the addition of the* Attr *element, signifying that input parameters can now also be attribute values;*
- $oVars$ *is the same as in definition 4.4.1.*

We use Attr *to signify "any literal attribute value", i.e. a string, boolean, integer or real attribute.*

The definitions for rules, system automaton and product automaton do not change.

5.3 Example usage

Using the modified rules and control language syntax, we can use attribute values straight from the control language. Properties such as the number of iterations to do on a certain rule (provided the rule keeps track of how many iterations it has done, by for instance reducing a counter) can thus be provided straight from the control language.

5.3.1 Actual attribute parameters

The simplest way to use attribute parameters is to type them in as literals. For instance, if we wanted to loop a certain rule five times, we would have a program like in listing 5.1, with the `setCounter` rule (figure 5.4) taking one input parameter which initializes a counter to a certain value and the `reduceCounter` rule (figure 5.5) reducing this counter by a number specified as another input parameter. We can change the initial value of the counter and the amount by which it is decreased from the control program, without touching the rules or the host graph. In both of the rules the parameters need to be specified as input-only, as explained in the previous section.

5.3.2 Formal attribute parameters

We can also get the value for the counter from some other rule rather than inputting it ourselves, as in listing 5.2. Here we first declare an integer variable, which we then initialize using the

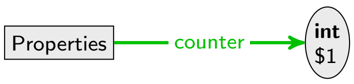


Figure 5.4: The setCounter rule initializes the counter to a given value

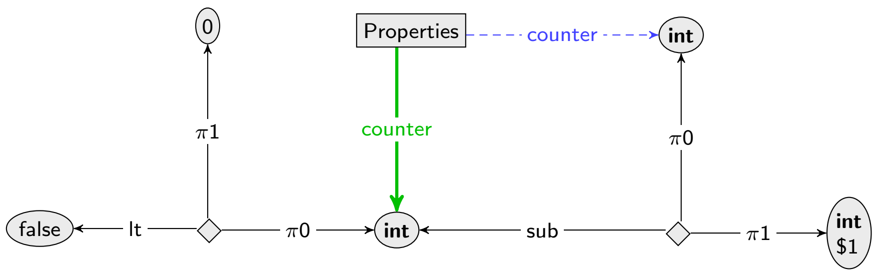


Figure 5.5: The reduceCounter reduces the value of the counter by a given value, as long as this would not reduce the counter to a value < 0

```
int loopCounter;

// some rules which modify the host graph, one of which
// introduces an integer attribute node somewhere

getCounter(out loopCounter);
setSecondCounter(loopCounter);

while(reduceCounter(1)) do {
  // some loop here
}
```

Listing 5.2: Example control program with attribute parameters

`getCounter` rule (figure 5.6). This rule matches some nodes, one of which is an integer attribute node the value of which will be saved in the `loopCounter` variable. We then pass this variable to the `setSecondCounter` rule (figure 5.7) as in the previous example.

Note that this time we did not have to specify the attribute parameter in `getCounter` to only be an input parameter, because it has an incoming edge in the left-hand side (and in fact we are using it as an output parameter, so Groove would have given us an error if we had specified it as `parin:`). While we can use this parameter as input, it would not do much except check that the *Properties*-node actually has a *counter*-edge to the supplied attribute node. In the `setSecondCounter` rule however, we do have to specify the attribute parameter with `parin:`, for the same reason as described in the previous example - it is not bound by the rule so it must be bound by the control program.

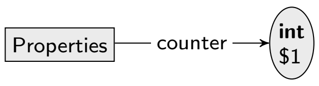


Figure 5.6: The getCounter rule takes the counter from the host graph

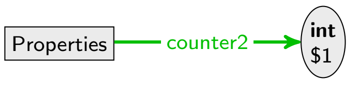


Figure 5.7: The setSecondCounter sets the value of a second counter value

Chapter 6

Implementation

The details of the implementation of the new features in GROOVE are explained in this chapter. We implemented and tested the input/output parameters and attribute parameters separately. Dividing the work like this, rather than implementing all the changes at once, allowed us to focus work on one thing, making testing easier. This chapter however, contains a description of the finished work.

The implementation consists of several parts, each with their own functionality, which when combined implement the new features:

- A control parser, syntax checker and determinism checker;
- An automaton builder;
- A system for deciding which rules to match and finding matches;
- A system to apply the rules and generate the new states.

In the following sections, we will discuss each of these parts individually.

6.1 Parser and checker

In order to generate a control automaton from a control expression, we have made a parser that turns text into an abstract syntax tree. Note that part of this functionality was already implemented by Tom Staijen.

This part consists of four sub-parts, each of which transforms or checks the tree, all implemented using ANTLR [10]:

- The lexer turns the textual representation into a stream of tokens;
- The parser takes a token stream and turns it into an abstract syntax tree;
- The syntax checker takes this syntax tree and checks it for valid syntax. This checks whether:
  - Whether variables are declared at most once;
  - Whether variables are declared before they are used;
  - Whether variables are initialized before they are used as input;
  - Whether variables are only assigned to once (see definition 4.6.1);
  - Whether variables are used in the scope they are declared in (or a lower scope).
- A determinism checker then takes the tree again and checks whether it describes a deterministic automaton as per definition 4.7.1.

After these steps have been performed, we have an abstract syntax tree which describes our control automaton. This tree must then be transformed into the actual automaton.

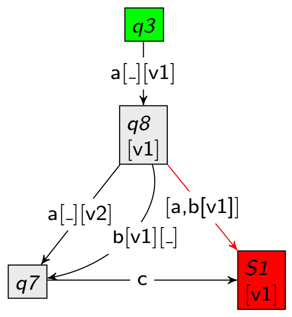


Figure 6.1: An intermediate control automaton

6.2 Automaton builder

When we are sure to have an AST which meets our requirements, we can transform it into a control automaton. For this, we once again take several steps:

1. We have one more ANTLR tree walker which takes the tree produced by the determinism checker and uses methods from the `AutomatonBuilder` class to generate an automaton;
2. The `AutomatonBuilder` then optimizes the automaton by performing the step described in definition 4.8.1, collapsing failure transitions with the 'real' transitions they precede.

In the first step, simple rule-calls are transformed into states and transitions with the name of the rule and any parameters on the label. More complex statements, like for instance `a(out v1); try { choice {a(out v2);} or {b(v1);} c; }` require more complex solutions (figure 6.1, the `[a,b[v1]]` label indicates that the *a*-rule does not have any matches and the *b*-rule do not have any matches with the input parameter list `[v1]`), though these are all combinations of the basic building blocks (in this case, a `try`, `choice`, some rule calls and two sequential compositions).

The second step is implemented as follows:

1. We construct a map which shows for every state in the control automaton, the states from which this state is reachable through only failure transitions, keeping track of the paths required to reach this state;
2. For every non-failure transition, we make transitions from all states which can reach the source state to the target state, preserving the original failures as negative guards;
3. We remove the failure transitions;
4. We remove the unreachable states (some states are now unreachable due to the previou step);
5. We mark states that used to be able to reach a success state with only a failure transition as conditional success states.

A class diagram (with only the relevant information for this report) of the control automaton structure built by the `AutomatonBuilder` can be found in figure 6.2. `ControlStates` keep track of which variables are initialized, so that we can later find the values for these variables (as product states only remember the values, not the names).

6.3 Match finder

For any state of the exploration, zero or more matches will be enabled for certain rules. Which rules these are is determined by the `ControlState` the simulation is in as well as the host graph (which contains the values for any variables that are relevant in the state). We use a state cache to keep track of a state and its transitions, called `RuleEvents`, with their targets in the LTS. Matching then occurs as follows:

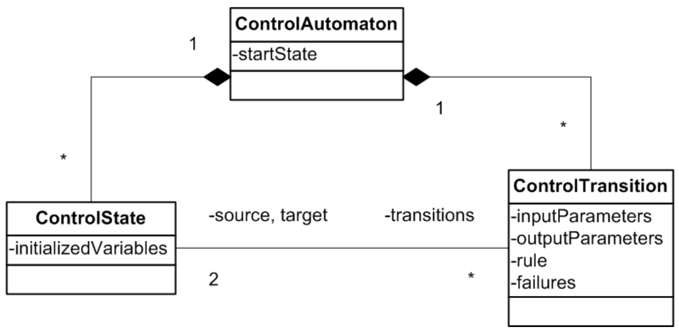


Figure 6.2: Class diagram for the control automaton structure

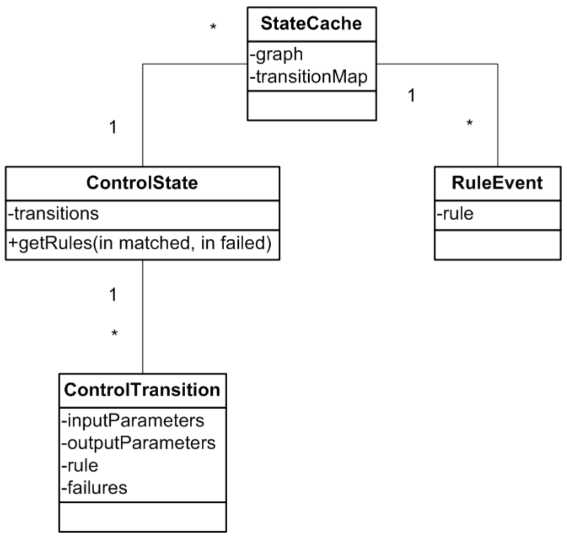


Figure 6.3: Class diagram for the matching structure

1. The state cache asks the control state which rules are to be scheduled, given that no rules have been matched yet and providing the values for the variables in the current graph state;

2. The control state returns a set of rules which can be applied without any failures and with the given variable values;

3. The state cache tries to match each of the rules and keeps track of which rules matched and which did not;

4. The state cache asks the control state which rules are to be scheduled, telling it which rules have already matched and which have not, also reporting with which input parameters these rules have matched and failed;

5. The control state returns a set of rules which should be scheduled based on the given failures (excluding rules which have already been matched).

Steps 3, 4 and 5 are repeated until step 5 returns an empty set. Using this strategy, we can ensure a minimal number of rules will be checked for matches. Another partial class diagram for this part can be found in figure 6.3.

6.4 Rule application

To get the matches for a rule in a given state, we must take care to apply the input parameters to these matches. We do this by constructing a partial match morphism out of the input parameters (in which attribute parameters always map to the same node and node parameters map to their values in the host graph) and then checking whether we can find one or more matches using this partial morphism as a base. For each of these matches we can then create a target state to add to the LTS - if it does not already exists in the LTS. These target states are created as follows:

- We apply the rule to the host graph, using the match morphism we found while checking for matches;
- If the source state had variables, we apply the comatch morphism to them to get the new values for these variables since nodes can get new identities through the rule application;
- If the transition has output-parameters, we find the nodes which these parameters map to by using the match-morphism to find them in the host graph, and we save these variable values in the target state.

We then end up with a target state which may be a new state, or may be a state that already exists in the LTS. To check whether this is the case, we have to know three things:

1. Whether the host graphs of both states are isomorphic;
2. Whether both states are 'in' the same `ControlState`;
3. Whether the values of the variables in both states point to identical nodes.

For the third step we use a method to get a canonical representation of a node, so that two nodes that are "identical" but do not have the same identity will still get the same value. This way we can compute whether two states are the same state modulo node identities.

If the target state is considered to be identical to a state already in the LTS, we do not add the target state and instead add a transition to the existing state. If the target state is considered to be a new state, we add it as a new state and add a transition to it.

6.5 Testing

Apart from manually trying to use the new features after every iteration, we wrote JUnit[1] tests cases to test the implementation. We tested the control automaton generation separately from the exploration of grammars, running these tests regularly to ensure new functionality was not breaking old functionality.

Control automaton generation To test the control automaton generation, we wrote a number of control programs using the various features of the language. We then let Groove generate a control automaton out of these and compared the resulting automata to automata we constructed by hand, checking for isomorphism. The source code for these tests can be found in the class `groove.test.ControlTest`.

Grammar exploration To test the exploration of grammars using control programs with variables, we wrote two more JUnit test cases; one for node parameters and one for attribute parameters. Inside these test cases we used a default grammar containing rules with one, two and three parameters and a number of control programs using each of the features we implemented. We then let Groove generate a control automaton and a state space out of these, and verified that the number of control states- and transitions as well as the number of LTS states- and transitions equalled the numbers we obtained by running the explorations manually (and verifying that the exploration was done correctly). The exploration tests can be found in the classes `groove.test.ControlVariablesTest` and `groove.test.ControlAttributeParametersTest`.

[1]See `http://www.junit.org`

6.6 Alternative implementation: encoding parameters in graph rules

As an alternative approach to implementing input and output parameters directly, we considered making copies of the called graph transformation rule with the parameters encoded into them. For example, a call like `r(out a);` would put a special node-label like *var_a* on the host graph where the first parameter was matched. The call would then be replaced by a normal rule without parameters, called `r_out_a;`.

Similarly, input parameters would be encoded as follows. A call like `r(a);` would be replaced by `r_a;`, with the rule *r_a* looking for a node label *var_a* and matching on that.

This approach has the benefit that the implementation of the matching and application of rules would not have to change. However, it would generate a large number of extra rules (one for each rule application with parameters in the control program) which need to be shown in the program. We would also have to somehow remove variables that go out of scope from the host graph, which in turn makes the control automaton more complex.

Selim Ciraci used an adaptation of this approach in his work[2], where he preprocessed the rules outside of Groove in order to achieve the same result. However, we have decided not to go with this as the amount of changes to the GUI and the control automaton and the amount of extra rules created outweigh the benefits of not having to adapt the matching and application of rules with parameters. Additionally, the changes to the host graph would be the opposite of what we were trying to do initially - which was to move complexity from the host graph and rules into the control program.

Chapter 7

Examples

This chapter contains examples of the new usage of the Groove control language, showing how the new features can be used to reduce the amount of control information in the graph transformation rules and the host graph.

7.1 AntWorld - end after n turns

As the AntWorld[20] case defines an infinite production system which simulates ants following predefined behavior to explore a grid and find food. As computers can not explore infinitely large transition systems, it is desirable to have an end-condition to prevent the LTS from growing larger than we want.

The way this is done in the current implementation is by way of having a simple rule that checks whether a set number of turns has passed and having the control program stop the exploration when this rule matches. An example of this rule can be found in figure 7.1 (this rule detects whether 10 turns have passed) and the relevant parts of the corresponding control program are in listing 7.1. Even though the rule is very simple, there is no way to make the exploration go to 5 or 20, or any other number of turns without adding extra rules and changing the control program.

With the extensions done in this project, we can change this situation by specifying a single rule (figure 7.2) and specifying the amount of turns to run for in the control program (see listing 7.2 for an example which once again ends after 10 turns). With this new combination of the `end` rule and the control program, we only have to change one value in the control program if we want to change the number of turns to simulate.

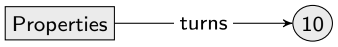


Figure 7.1: The rule for the AntWorld case which checks whether 10 turns have passed

```
until(end_10) do {
  // see if new ants should be created
  // perform ant moves / actions
  // see if grid should be expanded due to ants being
  //   at the outer edge of the grid
}
```

Listing 7.1: Control program for AntWorld which ends after the end_10 rule has been applied

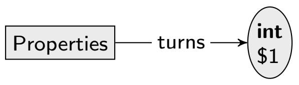


Figure 7.2: The rule for the AntWorld case which checks whether a specified number of turns have passed

```
until(end(10)) do {
  // see if new ants should be created
  // perform ant moves / actions
  // see if grid should be expanded due to ants being
  //   at the outer edge of the grid
}
```

Listing 7.2: Control program for AntWorld which ends after 10 turns

```
while (select_ant) do {
  try { take_food; } else {
    try { move_home; } else {
      try { drop_food; }
      try { move_to_pheromones; } else {
        move_random;
      }
    }
  }
  mark_ant_done;
}
```

Listing 7.3: Control program snippet to handle ant movements

7.2 AntWorld - Ant selection

Continuing from the AntWorld example, suppose we have a loop that selects an ant that has not moved yet in this turn, and then performs the appropriate action for this ant. The way we could do this without parameters is by adding a marking, or flag, to the selected ant and try to apply the rules that deal with handling ants to it.

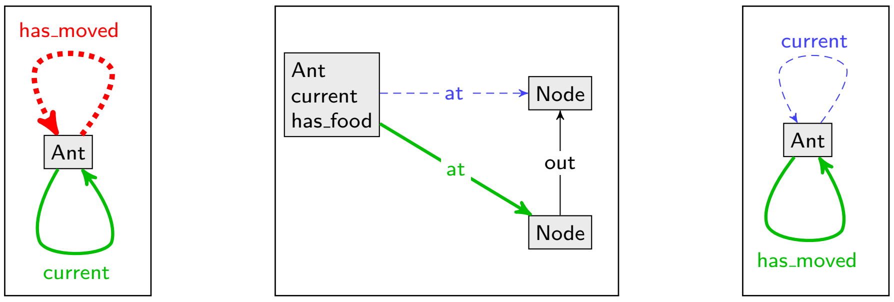
Consider the partial control program in listing 7.3. The control program tries to perform each of the specified actions and then falls back to a lower priority action if the tried action can not be applied. At the end, the ant is marked as having moved this turn (figure 7.3c). The way ant selection occurs is by adding a special label to one of the ant nodes, such as in figure 7.3a. The other rules then test for the presence of this label to ensure the same ant is always the one taking the actions (such as for instance the `move_home` rule in figure 7.3b).

The way we could write this control program with parameters is shown in listing 7.4. The new versions of the rules are in figure 7.4. In this new version we do not add the `current` label to our host graph, thus moving the control information to the control program. The difference is not big in this case, but it allows for more versatility. For instance, we can now use the individual rules from the loop outside the loop as well if we wanted to write a different control program - with the old rules this was not possible because the `current` edge would not be present.

7.3 Control flow generation

As an assignment for the Modelling and Analysis of Concurrent Systems 2[1] course, students were asked to write two rule system - one that allows generation of a control flow graph and another that allows 'execution' of this control flow. The execution of the control flow graph can be seen

[1]See `http://fmt.cs.utwente.nl/courses/macs2/`

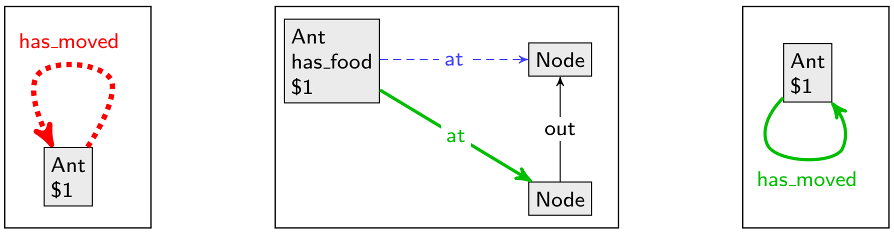


(a) select_ant (b) move_home (c) mark_ant_done

Figure 7.3: The ant movement rules without parameters

```
node current_ant;
while (select_ant(out current_ant)) do {
  try { take_food(current_ant); } else {
    try { move_home(current_ant); } else {
      try { drop_food(current_ant); }
      try { move_to_pheromones(current_ant); } else {
        move_random(current_ant);
      }
    }
  }
  mark_ant_done(current_ant);
}
```

Listing 7.4: Control program snippet to handle ant movements with control



(a) select_ant (b) move_home (c) mark_ant_done

Figure 7.4: The ant movement rules with parameters

as a stack machine; execution follows the `flow` labels on the graph and places information on a stack for some nodes, whereas other nodes consume information from the stack.

In this example we will focus on the rule system that generates the control flow graph. In all the pictures pertaining to this example, bold node labels are node types. The node labels `-Type` and `+Type` indicate that a rule changes the type of a node, removing the type with the `-` and adding the type with `+` in front of it.

Given the start state in figure 7.5, we can use several rules to modify the states to generate a control flow, eventually ending up with a representation of a computer program, such as the one in figure 7.6. The (pseudo-)code for this flow chart is in listing 7.5. We generate the control flow graph by sequentially specializing the `Elem` nodes into other types and then specializing those further. Some of these specialization rules add new `Elem` nodes, allowing the program to grow. A partial type graph for this grammar, containing only the node types relevant for this example, can be found in figure 7.7.

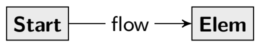


Figure 7.5: The start state for the control flow generation grammar

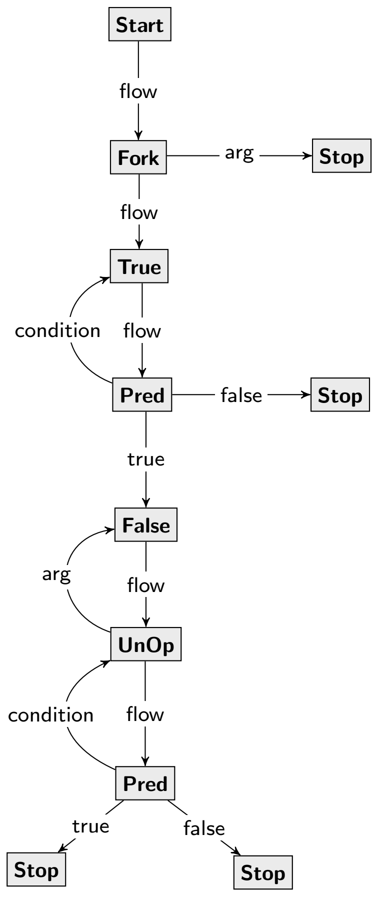


Figure 7.6: The target state for the control flow generation grammar

As the grammar describes an infinite state space, we need to use control (using a control program or by manually selecting matches) to get to the control flow graph that we want to have. We can generate our flowchart using the control program from listing 7.6. Each rule modifies the current control flow chart as follows:

- the `elem_proc` (figure 7.8a) rule adds a proc to the control flow;
- the `proc_stat` (figure 7.8b) rule specializes a proc into a statement;
- the `stat_fork` (figure 7.9) rule indicates a fork in the program, where a new thread should be created and both the `flow` and `arg` arrows are to be followed by a thread;
- the `elem_stop` (figure 7.8c) rule indicates that the flow should end at this node;
- the `elem_pred` (figure 7.10) rule replaces a program element with a predicate, adding a condition and new elements for whether the condition evaluates to true and false;
- the `cond_const` (figure 7.8d) rule indicates that this condition will be a constant (true or false);
- the `const_true` (figure 7.8e) and `const_false` rules give a constant node a value;

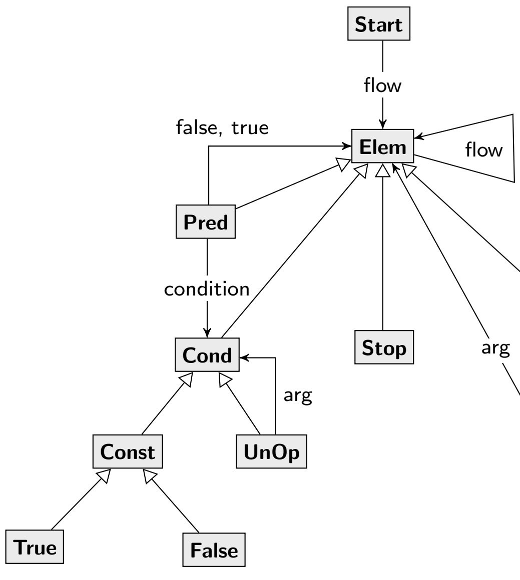


Figure 7.7: The partial type graph for the control flow graph grammar

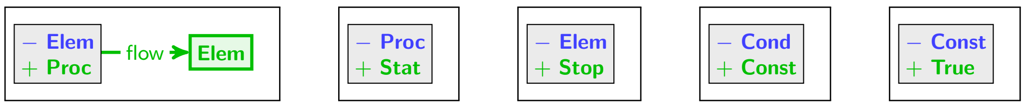
```
fork
{ exit; }
{
  if (true) {
    if (!false) {
      exit;
    } else {
      exit;
    }
  } else {
    exit;
  }
}
```

Listing 7.5: Pseudocode for the automaton from figure 7.6

```
elem_proc;
proc_stat;
stat_fork;
elem_stop;
elem_pred;
cond_const;
const_true;
elem_stop;
elem_pred;
cond_unop;
cond_const;
const_false;
elem_stop;
elem_stop;
```

Listing 7.6: First attempt at the control program to generate a control flow

- the `cond_unop` (figure 7.11) rule adds a unary operator to a condition node, effectively negating the condition. It also moves the condition edge from the containing predicate to the unary operator node instead of the condition node it was pointing to.



(a) elem_proc (b) proc_stat (c) elem_stop (d) cond_const (e) const_true

Figure 7.8: The simple rules for the control flow generation grammar

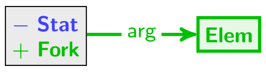


Figure 7.9: The stat_fork rule for the control flow generation grammar

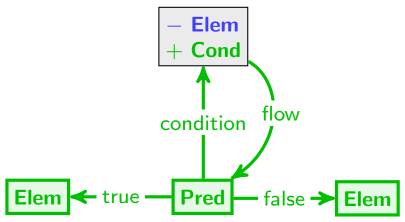


Figure 7.10: The elem_pred rule for the control flow generation grammar

Using the aforementioned control program generates the desired control flow, however it also generates three other control flow charts (such as the one in figure 7.12, which corresponds to listing 7.5 but with the true and false parts of the if statement on line 4 switched around). This is because some of the rules have several matches: the `elem_stop` rule on line 4 of the control program can match in two places, as can the one on line 8 (two matches for each of the on line 4 generated branches). This is because there are several `Elem` nodes in the host graph when Groove encounters this rule, and we have no way of telling it which one we want to match. This same effect could happen for any case where several matches exist for a rule. The way we have constructed our control program in this case ensures that it only happens twice, but had we for instance delayed the calls to `const_true` and `const_false` to the end, the first of these calls would also have two matches. The only feasible way to solve this without using parameters is to manually select the proper matches. Generating rules which add tokens to the host graph and other rules which consume them would work for small examples, but the larger the program we want to generate, the more rules have to be added - it would be easier to just draw the desired resulting graph instead.

With parameters however, this problem is easily solved. If rules which add `Elem` nodes are able

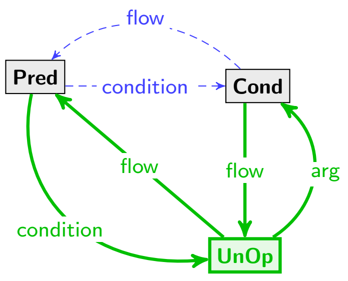


Figure 7.11: The cond_unop rule for the control flow generation grammar

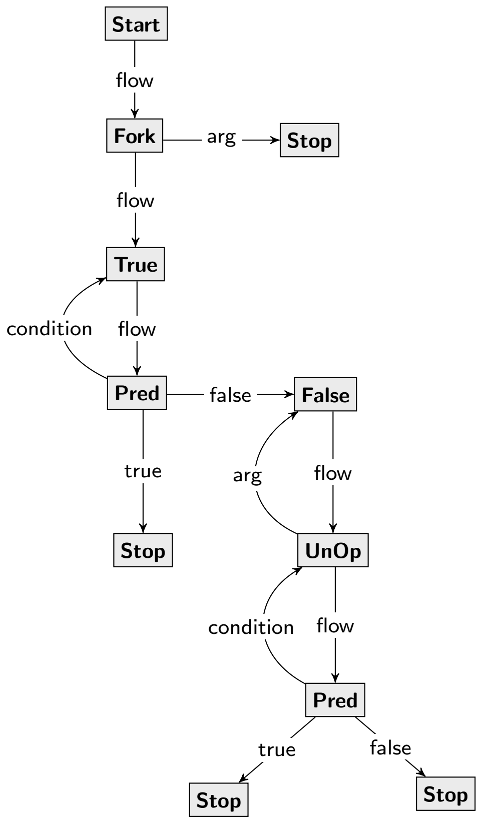
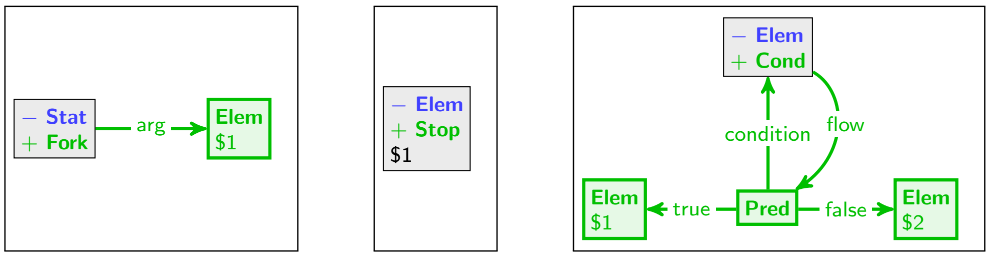


Figure 7.12: Another possible end state for the control flow grammar

to return these nodes as parameters, we can then tell rules that operate on an `Elem` node which one we want to match. For this example, the rules we have to change are `stat_fork`, `elem_stop` and `elem_pred`, though in a less contrived example it would make sense to add parameters to more rules.

The new rules can be viewed in figure 7.13 and the new control program is in listing 7.7. Note that we have chosen to explicitly define the order in which the `Stop` nodes on lines 14 and 15 are created. This was not required in this case to reach the same result, but in other cases it might have been. With the new control program, we have an entirely linear LTS with no choices between rule applications, thus we always end up in the same state - which is what we wanted in the first place.



(a) stat_fork (b) elem_stop (c) elem_pred

Figure 7.13: The replaced rules for the control flow grammar with parameters

```
node forkArg, predTrue, predFalse, pred2True, pred2False;
elem_proc;
proc_stat;
stat_fork(out forkArg);
elem_stop(forkArg);
elem_pred(out predTrue, out predFalse);
cond_const;
const_true;
elem_stop(predFalse);
elem_pred(out pred2True, out pred2False);
cond_unop;
cond_const;
const_false;
elem_stop(pred2True);
elem_stop(pred2False);
```

Listing 7.7: The control program which uses parameterized rules

```
function addStructuralFeature_ECLASS(){
  addConverterClass_ECLASS;
  implementCanConvert_ECLASS;
  checkConverted_ECLASS;
  addGraphNode_ECLASS;
  addStructuralFeatureIterator_ECLASS;
  addNameSet_ECLASS;
}
```

Listing 7.8: A control program from Selim Ciraci's thesis

7.4 Computer-supported design idiom verification

As mentioned in section 6.6, Selim Ciraci used a form of parameters in his thesis work [2]. He wrote rulesystems to verify design idioms [16], which we will not go into here but in which several rules use string constants which are to be specified after the rule has been specified. These rules are quite complex, but contain single string attributes which have to match certain values from a control program. The approach Mister Ciraci used to tackle this in his thesis was to use a special syntax in the control program and mark the attributes with an `@`-prefix. He then used a self-written preprocessor to make copies of the rules with the literal attribute values from the control program for these attribute nodes, replacing the `@`-attributes.

An example control program from this approach can be found in listing 7.8. In this program, `ECLASS` stands for the part that is to replace the `@`-attributes in the rules that are called. An example of one of these rules can be found in figure 7.14. As can be seen, these rules can get quite big, but the important part for this example can be seen in the upper left corner of the figure. The attribute node with the label `"@ClassName"` will need to be adapted for every copy made of this rule, to reflect the value from the call in the control program.

As we have to make copies of each of the rules that contains a variable, the rulesystem quickly becomes cluttered with nearly-identical rules. To alleviate this we can use attribute parameters in the rules and then use the literal attribute parameter feature discussed in chapter 5 to change these values from just the control program. The control program could then be rewritten like in listing 7.9, with the adapted rule in figure 7.15.

One thing to note here however, is that we would still need to write a new `addStructuralFeature_NAME` function for every value of the parameter we wish to use, as we do not currently support parameters in functions. Working out and implementing the ideas from section 3.2, we would be able to make this example even better, by using a function parameter to allow us to only have to specify the value once. An example of this can be found in listing 7.10. Regardless, replacing the literal node with an attribute parameter saves us from having to copy the rules for each valuation of the parameters, thus resulting in a smaller rulesystem which

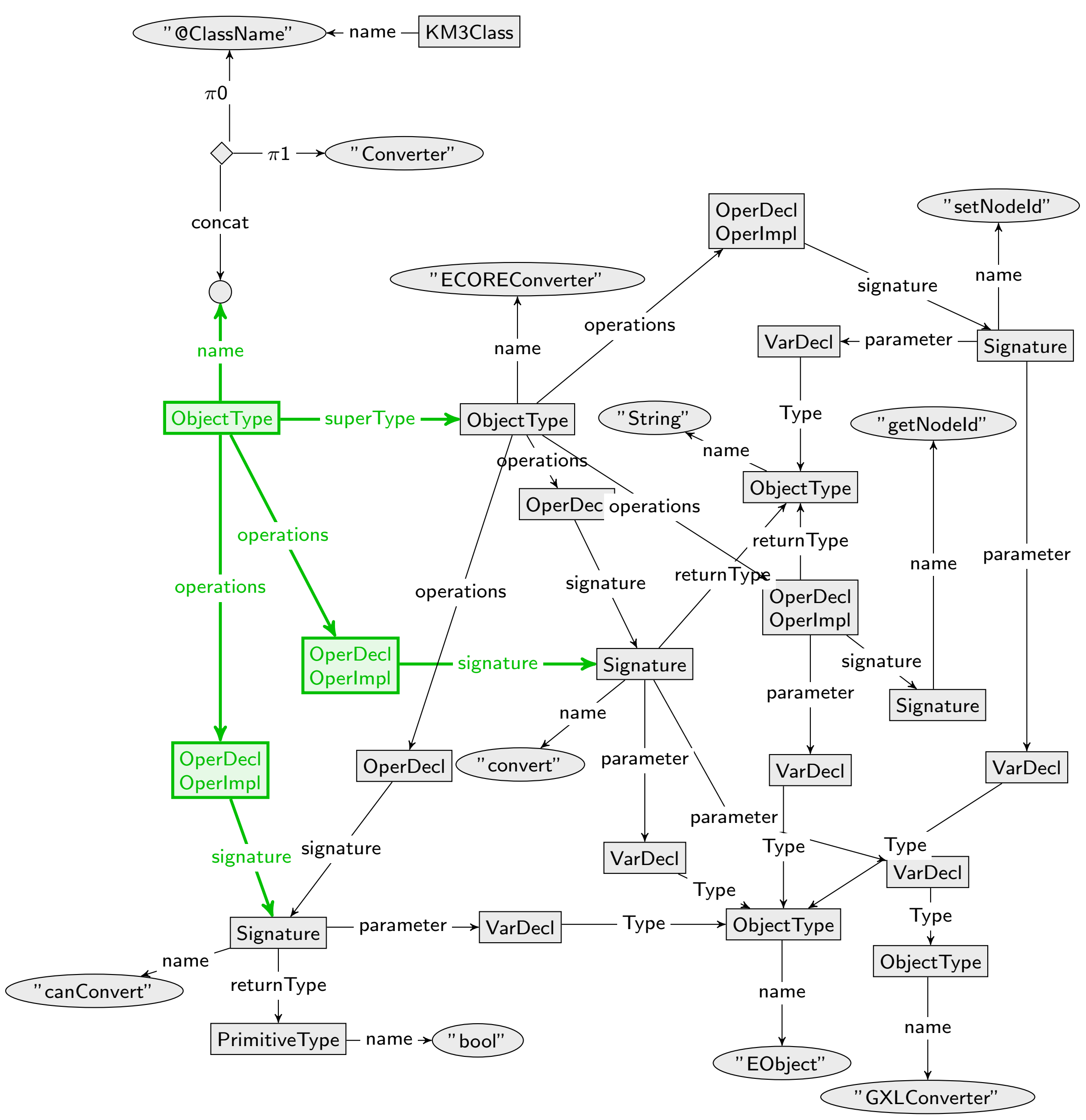


Figure 7.14: The addConverterClass rule from the idiom verification ruleset

```
function addStructuralFeature_ECLASS(){
   addConverterClass("ECLASS");
   implementCanConvert("ECLASS");
   checkConverted("ECLASS");
   addGraphNode("ECLASS");
   addStructuralFeatureIterator("ECLASS");
   addNameSet("ECLASS");
}
```

Listing 7.9: The same control program with attribute parameters

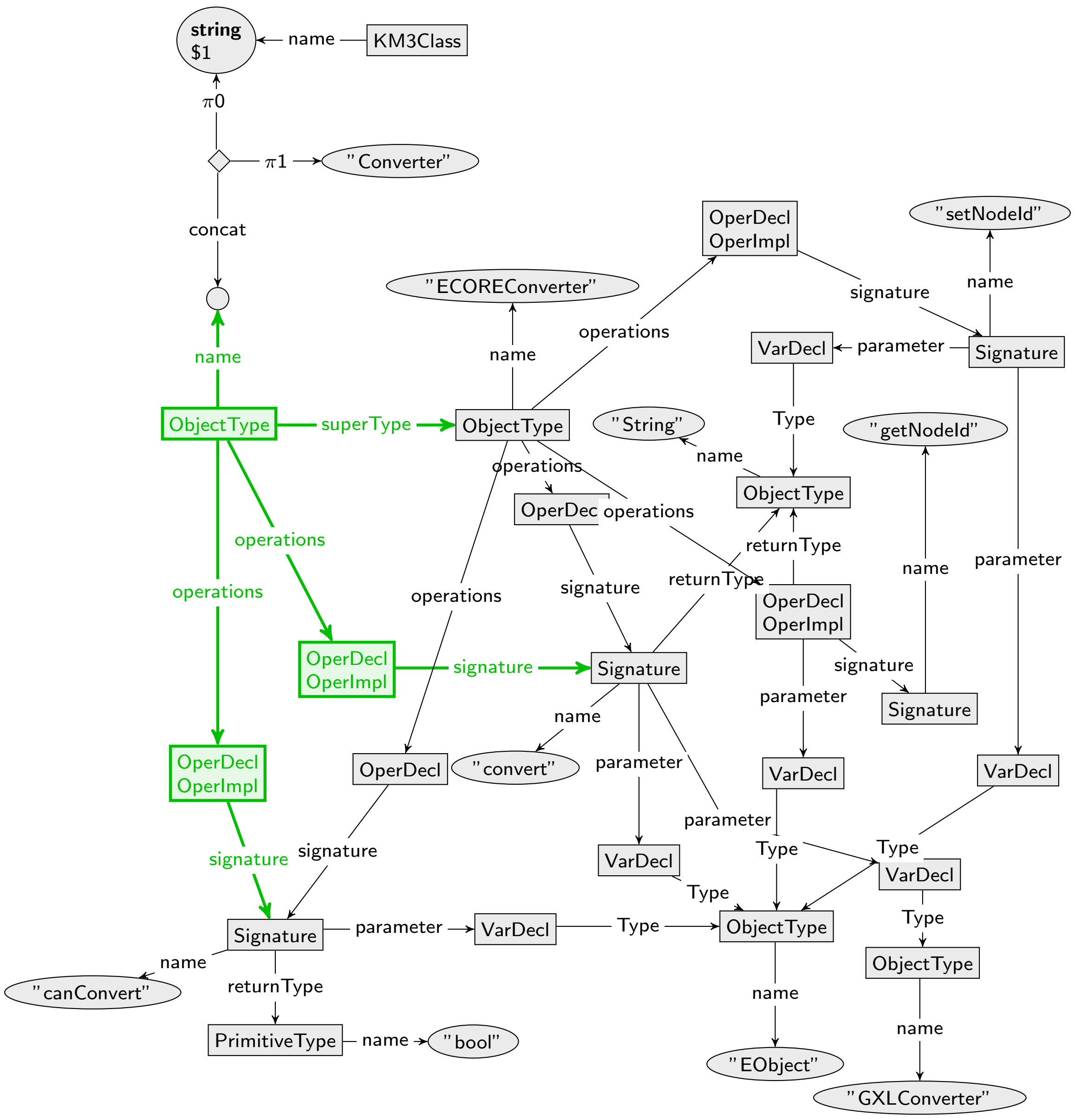


Figure 7.15: The addConverterClass rule from the idiom verification ruleset, with attribute parameter

is easier to understand.

```
function addStructuralFeature(string value){
  addConverterClass(value);
  implementCanConvert(value);
  checkConverted(value);
  addGraphNode(value);
  addStructuralFeatureIterator(value);
  addNameSet(value);
}
```

Listing 7.10: The same control program with attribute and function parameters

Chapter 8

Conclusions, related work and future research

This chapter contains the final word on this research. We will discuss whether we have attained the goals we set at the start and discuss ways to continue. We also have a comparison of other graph transformation tools, focusing on how they handle control.

8.1 Conclusions

Looking back on this research project, we can now decide what the answer to our research question (*"Can we remove control information from the graph transformation rules and host graph and move it into the control program by using parameterised rules?"*) is.

During this project, we first defined a number of possible extensions to Groove which could be used to answer this question, then we worked two of these extensions out mathematically. By far the most work was put in the first extension, basic input/output parameters (chapter 4), due to the complications involved with nondeterminism. It is because of this that we eventually decided to disallow nondeterminism altogether. We then built on this extension with input/output parameters by mathematically explaining how we can use attributes as input and output in graph transformation rules and how we can input these as literal values from a control program, in chapter 5.

With the exception of allowing nondeterministic control expressions, we have implemented the ideas from chapter 4 and the ideas from chapter 5 and used them in the examples from chapter 7. These examples show that with some care, we can reduce the amount of control information we have to write in graph transformation rules and our host graph by moving it to the control program.

The AntWorld examples show that we can now do with one rule what previously required several nearly identical rules, and that we can reduce the amount of context-information from the host-graph. The former rules all did the same thing with one minor modification - an attribute node has a different value in each of the rules. We have condensed all these rules into one by requiring the user to input the value for the attribute node from the control program instead.

The control flow generation example from section 7.3 shows that, in addition to reducing the amount of rules we need and the information we store in the host graph, we can actually make our grammars more expressive. While the results from this example (a fully linear LTS) could have been achieved without parameters, we would have had to add several one-use rules to make it possible. For bigger examples the amount of rules to add increases even more, making a solution without parameters unfeasible.

The example from section 7.4 finally shows us that not only do our extensions have a benefit in contrived examples, but also in real-world cases, as the rule and the control program presented in this example were taken from a case study by Selim Ciracy, which is presented in his PhD thesis [2].

Each of the examples shows that our extensions of the control language makes grammars easier to work with and makes the rules and graphs used more clear. We have done this without adversely affecting existing grammars - backwards compatibility is maintained, so grammars which were written before our extensions were made will still work; some grammars may be optimized using the new functionality however.

To conclude, while we have made some concessions as to which part of the theory we worked out was eventually implemented, we think we have made a worthwhile contribution to Groove, making it easier to express certain things in the control language that would otherwise involve extra work in the graph transformation rules and host graph.

8.2 Related work

Groove is not the only graph transformation tool available; several other groups are working on their own implementations of the graph transformation formalism. This section contains a survey of some of the other graph transformation tools out there, focusing on what types of control structures they support. We have obtained this information from a case study paper[14] about graph transformation tools, a paper on practical applications of Groove [6] and the information each of these tools presents publicly on the web.

8.2.1 VIATRA2

While VIATRA2 (VIsual Automated model TRAnsformations)[1] is officially a model transformation tool rather than a graph transformation tool, the models are expressed as graphs (or trees); as such it can be used for graph transformation as well.

In VIATRA, models are expressed in the VIATRA2 Model Space, which is a model management framework more expressive than MOF[8] and EMF[5]. It extends EMF by allowing things like navigating references in both directions and enumerating all instances of a metamodel element. Currently, there are importer plugins for UML2, BPEL and other model descriptions.

The transformation language consists of declarative and imperative features and is built upon graph transformation and abstract state machines. VIATRA2 uses an advanced imperative control language to control the rule scheduling.

8.2.2 FuJaBa

FuJaBa[9], or *From Uml to Java And Back Again*, is a tool developed originally by the University of Paderborn in 1997. In 2002 it has been redesigned as a plugin system, with Eclipse integration added in 2006. FuJaBa is being used extensively for educational purposes, both at the University of Kassel and in highschool courses for beginners.

FuJaBa's core modeling features consist of UML class diagrams and story diagrams, which are a combination of UML activity diagrams with the Story Pattern graph transformation language. The graph transformation rules look like UML class diagrams with transformation patterns similar to those in Groove.

Control in FuJaBa is done through the story diagrams, which link rules together by making transitions marked `[success]` and `[failure]`, indicating which rule to apply based on these conditions.

8.2.3 GReAT

GReAT[17], or *Graph Rewriting And Transformation*, is a tool for building model transformations using graph transformation techniques. Transformations are built by first specifying the metamodels of the input and target models, through UML class-diagrams.

Transformation is then specified by writing graph transformation rules which rewrite a part of the input model into a part of the target model. Rules consist of a pattern, a guard and a set of actions that modify the model. Matches are computed starting from specific nodes, called pivots, in order to limit the search. These pivots can be passed down to subsequent rules; as such the control flow is partially established inside the rules.

The rule execution is written by sequencing these rules, with some control structures such as conditionals and loops. The “output pivots” of one rule are the “input pivots” of the next rule, with the initial input pivots being selected manually from the input model. This type of control is called a dataflow paradigm.

8.2.4 AGG

AGG[7] (Attributed Graph Grammar System) is a tool developed by the graph grammar group at the TU Berlin. It supports graph transformations with rules being explicitly written in separate graphs: the left-hand side, the right-hand side and possible negative application conditions. As such, morphisms between these will also have to be specified, but the tool has mechanisms in place to assist the user with this by guessing the correct morphism.

The main difference though, is that in AGG most things can be attributed with Java language features. Graphs may be attributed with Java objects and types, and rules may be attributed by Java boolean conditions to indicate preconditions as well as Java expressions which will be evaluated during the application of rules.

AGG supports control of rule scheduling through the use of global rule-priorities. The user can select which rules have precedence over others, in the same way as Groove supports this.

8.2.5 PROGRES

PROGRES[11] (PROgrammed Graph REwriting Systems) has been under development since the late 1980s, making it one of the eldest graph rewriting tools. It can transform directed, attributed, node and edge labeled graphs, transforming these with two types of rules: *simple* and *combined*.

Simple rules are similar to graph transformation rules in Groove, consisting of a left-hand side, a right-hand side. The left-hand side may contain negative edges and nodes as well as set-valued nodes (akin to quantified nodes in Groove). Combined rules on the other hand consist of multiple rules with a textual control structure, similar to Groove’s control language.

8.3 Future work

We have added functionality to Groove in this project which allows us to move complexity from the rules and host graph to the control program. Still, there are more ways conceivable of achieving even more decoupling, or otherwise adding functionality to the control language that it does not currently have.

The first idea would be to extend the input and output parameters to functions. Functions are currently in-lined; that is, the call to a function is essentially replaced with the statements in that function. We would need to find a suitable semantics for function calls with parameters and implement it, this extension should not be too difficult to do as most of the functionality for scopes, inuput and output parameters already exists elsewhere. The example from section 7.4 would benefit from this functionality as it would allow us to only have to enter values once instead of in several calls.

Secondly, we could add functionality to the control language which is not directly visible in the host graph. For instance, we could allow manipulation of attributes in the control language directly, instead of only letting the user input them. Manipulating an attribute in a graph transformation rule (for instance to add 1 to an integer) requires several nodes, but in the control language it could be as simple as `var++;`. It would allow us to keep track of attribute values in the control program instead of the host graph; for instance, we could keep track of the number of turns that have passed in a game simulation without placing this information in the host graph. As such statements would not show up on the LTS at all, this might be a dangerous development, but when used properly it could add value.

As we can now change attribute values directly in the control program we can modify parameters easily, but we can not do this when using the command-line LTS generator of Groove since this does not have an editor (though we could of course use a text editor to edit the control program). In the simulator we can choose from one of several saved control programs, but the generator does not currently allow us to do this; it only uses the control program we selected last in the simulator. One way to solve this would be to accept a command-line argument specifying the name of the control program to be used.

Another way to solve it, which could also be usable in the simulator, is to add special markings in the control program for "run-time attribute parameters", much like the markings in the graph transformation rules are used as control parameters. We could then write control statements such as `until( end(${how many turns?}); ) do { // loop }`, and provide a number for this value in the command-line arguments. The question we entered could be used by the simulator to have Groove ask the user for a value (rather than "enter an integer" it would specify what the integer the user enters would be used for). For the generator it could be used to generate a "how to use", i.e. a listing of all arguments and their meanings.

Lastly, while we have introduced some restrictions in chapter 4, it might be worth some attention to try and alleviate these. For instance, the restriction that each variable can only be used as output once could be solved by allowing variables to be used as output as many times as the user likes, and then converting the resulting automaton to static single assignment form using the methods described in [4] (using the Φ-function), thus allowing for a more convenient way of specifying control programs. The other restriction we introduced, where nondeterminism in control programs is no longer allowed, could also be removed if we implement the ideas that, for a large part, we already worked out in the aforementioned chapter. This would likely require more implementation work than has been done in this project, so an analysis of the possible benefits would first have to be conducted.

Bibliography

[1] Budapest University of Technology and Economics, HUN and OptXware Research and Development LLC, HUN. Viatra2 - eclipsepedia, September 2009. `http://wiki.eclipse.org/VIATRA2`.

[2] Selim Ciraci. *Graph Based Verification of Software Evolution Requirements*. PhD thesis, University of Twente, December 2009.

[3] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Lowe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. *Handbook of Graph Grammars and Computing by Graph Transformation*, 1:163–245, 1997.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[5] The Eclipse Foundation. Eclipse Modeling Framework, 2009. `http://www.eclipse.org/modeling/emf/`.

[6] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and Analysis Using GROOVE. Technical Report TR-CTIT-10-18, Enschede, April 2010.

[7] TU Berlin Graph grammar group. The AGG Homepage, 2009. `http://user.cs.tu-berlin.de/~gragra/agg/`.

[8] Object Management Group. OMG's MetaObject Facility, January 2009. `http://www.omg.org/mof/`.

[9] University of Paderborn. Fujaba Tool Suite, 2009. `http://www.fujaba.de/about-fujaba.html`.

[10] Terence Parr. Antlr parser generator, 2010. `http://www.antlr.org`.

[11] Ulrike Ranger and Erhard Weinell. The graph rewriting language environment PROGRES. *Lecture Notes in Computer Science*, 5088:575–576, 2008.

[12] A. Rensink. The GROOVE simulator: A tool for state space generation. *Lecture Notes in Computer Science*, 3062:479–485, 2004.

[13] A. Rensink and T. Staijen. Controlled Rule Application using Failure Automata. june 2009.

[14] Arend Rensink, Alexander Dotor, Claudia Ermel, Stefan Jurack, Ole Kniemeyer, Juan de Lara, Sonja Maier, Tom Staijen, and Albert Zündorf. Ludo: A case study for graph transformation tools. *Lecture Notes in Computer Science*, 5088:493–513, 2008.

[15] Arend Rensink and Jan-Hendrik Kuperus. Repotting the geraniums: On nested graph transformation rules. In A. Boronat and R. Heckel, editors, *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques, GT-VMT 2009*, volume 18 of *Electronic Comminucations of the EASST*. EASST, 2009.

[16] Mary Shaw. Heterogeneous design idioms for software architecture. In *IWSSD '91: Proceedings of the 6th international workshop on Software specification and design*, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[17] Institute for Software Integrated Systems Vanderbilt University. Graph Rewriting And Transformation, 2008. `http://www.isis.vanderbilt.edu/tools/GReAT`.

[18] Wikipedia. Ludo (board game) — wikipedia, the free encyclopedia, 2010. [Online; accessed 21-May-2010].

[19] Wikipedia. Pac-man — wikipedia, the free encyclopedia, 2010. [Online; accessed 5-May-2010].

[20] Albert Zündorf. Antworld, 2008. `http://www.se.eecs.uni-kassel.de/~fujabawiki/index.php/AntWorld` [Online; accessed 17-May-2010].