Controlled Rule Application using Failure Automata

Arend Rensink[2] and Tom Staijen[1]

[1] Formal Methods and Tools Group, University of Twente, The Netherlands,
rensink@cs.utwente.nl
[2] Software Engineering Group, University of Twente, The Netherlands,
staijen@cs.utwente.nl

Abstract. Pure rule-based systems typically consist of a single, unstructured set of rules. The behaviour of such systems is that all rules are applicable in every state. Rules can then only be forced into a certain order of application by adding special elements to the states, which are tested for within the rules. In other words, control over rule applicability is not explicit but has to be encoded in the state, which is bad for understandability and maintainability.
In this paper, we study so-called *control automata*, which can be added on top of pure rule systems. The resulting behaviour is defined as the product of the original state space and the control automaton. Control automata include so-called failure transitions, representing the observation of the non-applicability of one or more rules. The result is a reactive semantics for control expressions, which extends the usual input-output semantics in a consistent manner.
Control automata may introduce artificial non-determinism into the behaviour, which is an undesirable effect. We introduce *guarded control automata* to get rid of this effect, and we define a semantics-preserving transformation from ordinary control automata to guarded ones.

1 Introduction

The use of rule-based specification and programming languages to model complex systems is a commonly accepted approach in the field of computer science. In such rule-based systems, such as term-rewrite systems, Petri Nets and graph rewrite systems, typically applying a rule in a state typically results in another state, and all rules are scheduled — allowed to be matched and applied — in every state at all times. The result of this process is the full state space

The method of choice of the authors is graph transformation-based specification and programming. In this context, extending the rule system with a mechanism for controlling rule application is very popular. Such constructs, often called *control expressions* can increase the ease of specification in a rule-based language. The results of this paper, however, are not restricted to graph transformation systems.

One of the main advantages of using explicit control expressions, is that it reduces the amount of control information required in the states and rules. Such

information quickly complicates the comprehensibility of the entire rule-system, and introduces hidden dependencies between rules. Explicit control expressions, specified, on the other hand, reduce such control information and provides an explicit view on these dependencies.

Typically, control expressions in the field of graph transformations are given an *input-output* semantics, motivated by an interest in the transformational behaviour of a system. In contrast, we use graph grammars to generate a *reactive* view on the system. Graph grammars are used to generate the full state space of the system, which is used for verification by, for instance, model checking.

In this work, we introduce a control mechanism with a reactive semantics. We define so-called *control automata*, which can be added on top of pure rule systems. The resulting behaviour is defined as the product of the original state space and the control automaton. Control automata can include transitions that specify the scheduling of a rule, and so-called failure-transitions that describe the observation of a set of rules being inapplicable. A control language is defined for the purpose of specifying such control automata in a simple and intuitive manner. Because the product with ordinary control automata can introduce unwanted non-determinism, we also define a variation called *guarded control automata*, which do not have this undesirable effect.

In the next Section, we start by defining control automata and show the result of combining such an automaton with system automata: automaton representations of rule systems. In Section 3 we present a control language, and show how programs written in this language can be translated to control automata. In Section 4 we present guarded control automata, and we prove them to be equivalent to the normal automata. Finally, in Section 5 we present our contribution, implementation & scalability issues, related work, and future work.

2 Rule Systems and Automata

Since this paper is placed in the general context of rule systems, we first define our conception of such systems.

Definition 1 (rule system). *A rule system is a set of rules,* Rule*, which act on a universe of data structures,* Data*, in a manner captured by a partial* derivation *function* $\delta : \mathsf{Data} \times \mathsf{Rule} \times \mathsf{Id} \rightharpoonup \mathsf{Data}$*, where* Id *is a set of* application identifiers*.*

The fact that $\delta(d, n, i) = d'$, which we will henceforth denote $d \xrightarrow{r,i} d'$, expresses that rule n can be applied to structure d, resulting in a new structure d'. The identifier i provides information on how n was applied precisely; this has the effect of making the result d' unambiguous.

Without going into details, we note that this definition encompasses a wide spectrum of systems, including Turing machines, Petri nets, and various kinds of rewrite systems.

The meaning of a rule-based system is usually taken to be the *normal forms* reachable from a given input structure d; that is, those structures $d' \in \mathsf{Data}$ such that there exists a chain of derivations $d \xrightarrow{r_1,i_1} \xrightarrow{r_2,i_2} \cdots \xrightarrow{r_n,i_n} d'$ and there

are no further derivations possible from d'. In other words, one is interested in the *transformational* or *input-output* behaviour of the rule system. In this paper, on the other hand, we consider the *temporal* or *reactive* behaviour of rule systems, which can only be captured by taking intermediate steps into account. To formalise the reactive behaviour, we use *automata*. For the sake of simplicity, in this paper we identify rules with their names; hence, we will use rules as (part of) transition labels. Throughout the paper, we will use Rule to stand for the rule system under consideration, with associated sets Data of data structures and Id of application identifiers.

2.1 Automata

We distinguish system automata and control automata, with the same structure.

Definition 2 (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 or may not 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.*

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 $\ell \neq \lambda$.*

Intuitively, a success state is a state in which it is correct to halt execution. The special symbol λ stands for an *invisible* step. We use the following notations:

$$\begin{aligned} q \xrightarrow{\ell_1 \cdots \ell_n} q' \quad &:\Leftrightarrow \quad q \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_n} q' \\ q \overset{\epsilon}{\Longrightarrow} q' \quad &:\Leftrightarrow \quad \exists n : q \xrightarrow{\lambda^n} q' \\ q \overset{a_1 \cdots a_n}{\Longrightarrow} q' \quad &:\Leftrightarrow \quad q \overset{\epsilon}{\Longrightarrow} \xrightarrow{a_1} \cdots \overset{\epsilon}{\Longrightarrow} \xrightarrow{a_n} q' \quad . \end{aligned}$$

Moreover, we use $q \xrightarrow{a}$ to denote $\exists q' : q \xrightarrow{a} q'$, etc. Finally, we define the *language* of an automaton as the set of all traces, with a distinguished subset of *successful* traces; i.e., for all $\mathcal{A} \in \mathsf{Aut}$:

$$\mathcal{L}(\mathcal{A}) = (\mathcal{T}(\mathcal{A}), \mathcal{T}^{\surd}(\mathcal{A})), \text{ where } \begin{array}{ll} \mathcal{T}(\mathcal{A}) & = \{w \in (\Sigma \setminus \lambda)^* \mid q_0 \overset{w}{\Longrightarrow} \} \\ \mathcal{T}^{\surd}(\mathcal{A}) & = \{w \in (\Sigma \setminus \lambda)^* \mid q_0 \overset{w}{\Longrightarrow} \overset{\epsilon}{\Longrightarrow} q' \in S\} \end{array}$$

It follows from standard theory that every language (to be precise, every pair $(\mathcal{T}, \mathcal{T}^{\surd})$ with $\mathcal{T} \subseteq (\Sigma \setminus \lambda)^*$ non-empty and prefix-closed and $\mathcal{T}^{\surd} \subseteq \mathcal{T}$) is uniquely (up to isomorphism) represented by a deterministic automaton.

System Automata. A system automaton is essentially a state-transition system describing the step-by-step derivations of a rule system.

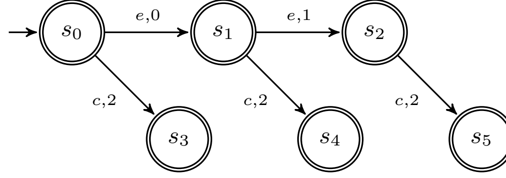


Fig. 1. Example System Automaton

Definition 3 (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 data structure* $d_q \in \mathsf{Data}$*, satisfying the following consistency properties:*

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

For arbitrary $d \in \mathsf{Data}$*, the* free automaton $\mathcal{A}_d$ *is a system automaton with data structures as states and derivations as transitions, which is the smallest such that* $q_0 = d$*, and for all* $d' \in Q$ *and* $\delta(d', n, i) = d''$*,* $d'' \in Q$ *and* $d' \xrightarrow{n,i} d''$*.*

The set of system automata is denoted SAut. The (r, i)-labelled transitions are essentially rule applications, whereas in a λ-transition no data transformation occurs. Given that states are data structures combined with some extra information, a λ-transition represents an (undefined) change in this extra information. The free automaton is the result of uncontrolled rule application at every state.

Example 1. Figure 1 shows a system automaton based on two rules:

e : This rule represents a passenger *entering* a train.
c : This rule represents the *closing* of the train door.

This simple process is shown for a start state where two passengers want to enter the train. The e-rule can be applied twice. The door can be closed once, regardless of the passengers being inside or outside the train.

Determinism. We follow standard automata theory in equating all automata with their sets of traces; or in other words, every automaton is considered to be essentially the same as its determinisation (according to the standard powerset construction). For system automata, this is justified because their labels are enriched so that they do not only contain rule names but also application identifiers. Of these, only the rule names are typically observable; for instance, verification methods such as model checking only take note of the rule name part of the labels.

If we would project all $\mathsf{Rule} \times \mathsf{Id}$-labels of a system aytomaton onto their first components, the resulting automaton would in general be non-deterministic; but this type of non-determinism can *not* be resolved without changing the meaning of the automaton. This implies that this projection is not well-defined modulo trace equivalence of the automata. For instance, in Fig. 2, automata (b) and (c)

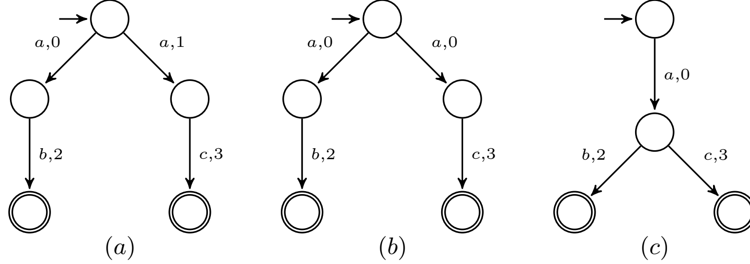


Fig. 2. Different types of non-determinism

are language equivalent, whereas (a) is different; however, after projection onto the rule names, (a) and (b) appear model essentially the same (viz., isomorphic) behavious whereas (c) appears to be different.

To avoid this type of confusion, we prefer to work only with deterministic system automata. Obviously, this can be achieved by determinising automata whenever necessary. However, this might not be the best technique, as determinisation can be exponential in the size of the automaton, and system automata are likely to be very large. Part of the contribution of this paper is therefore a technique to avoid generating non-deterministic system automata altogether.

Control Automata. Control automata are automata that can express on the one hand the application of a rule and on the other the observation that a given set of rules cannot be applied. The latter is called a *failure.* The set of all possible failures is given by $\mathtt{Fail} = 2^{\mathsf{Rule}}$.

Definition 4 (control automaton). *A* control automaton *is an automaton where* $\Sigma = \mathsf{Rule} \cup \mathtt{Fail}$*, such that:*

- *For all* $q \in Q$, $q \xrightarrow{F}$ *with* $F \in \mathtt{Fail}$ *only if* $\forall a \in F : q \stackrel{a}{\not\to}$.
- *For all* $q \in S$, *there is no transition* $q \xrightarrow{a}$ *or* $q \xrightarrow{F}$.

The class of control automata is denoted CAut. Note that the empty failure represents the fact that all rules in the empty set are inapplicable, which is vacuously true; hence, an empty failure transition is effectively a λ-transition.

Example 2. Figure 3 shows a control automaton in which rule e is scheduled as long as it is possible, after which c is scheduled. In the setting of Example 1, the door is closed only when no more passengers want to enter the train. Here, $c_0 \xrightarrow{[e]} c_2$ denotes a failure transition with label $\{e\}$.

2.2 Combining System Behaviour and Control

The idea of a failure as the observation of a set of rules being inapplicable is given a meaning by defining the product of control and system automaton. This results in another system automaton, where states are tuples of a system state and a control state.

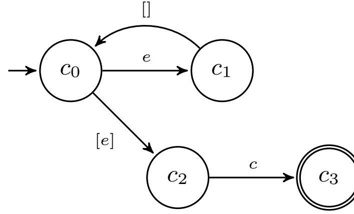


Fig. 3. Example Control Automaton

Definition 5 (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:*

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

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

The control automaton constrains the system automaton on the name-component of its traces. In particular, in a combination of system state and a certain control state, a failure transition from that control state can be taken when for none of the names in the failure set an outgoing rule application exists in the system state.

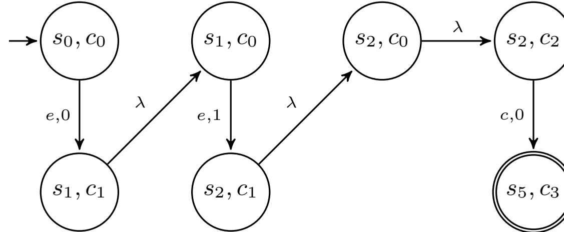


Fig. 4. Example Product System Automaton

For example, Figure 4 shows the system automaton that is the product of the system and control automaton in Figures 1 and 3.

3 Control Language

For the convenient specification of control automata, we propose to use a control language along the lines of what has previously been proposed in, e.g., [7]. We will use the following grammar:

$$P := \mathsf{rule} \;|\; \textbf{true} \;|\; P_1 | P_2 \;|\; P_1 ; P_2 \;|\; P* \;|\; \textbf{alap}\; P \;|\; \textbf{try}\; P_1 \;\textbf{else}\; P_2$$

These constructs have the following intuitive meaning:

- 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;
- **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.

Traditionally (e.g., in [7]), the effect of such control programs is defined in terms of the resulting input-output behaviour for the rule system at hand. Formally, this is captured by a function $[\![-]\!]_{\mathsf{io}} : \mathsf{Lang} \to \mathsf{Data} \times \mathsf{Data}$. For instance, for some of the operators the defining clauses are as follows:

$$\begin{aligned}
[\![\mathsf{rule}]\!]_{\mathsf{io}} &= \{(d, d') \mid \exists r, i : d \xrightarrow{r,i} d'\} \\
[\![P_1; P_2]\!]_{\mathsf{io}} &= \{(d, d') \mid (d, d'') \in [\![P_1]\!]_{\mathsf{io}}, (d'', d') \in [\![P_2]\!]_{\mathsf{io}}\} \\
[\![P*]\!]_{\mathsf{io}} &= \{(d, d) \mid d \in \mathsf{Data}\} \cup \{(d_0, d_n) \mid (d_0, d_1), \ldots, (d_{n-1}, d_n) \in [\![P]\!]_{\mathsf{io}}\} \\
[\![\textbf{alap } P]\!]_{\mathsf{io}} &= \{(d, d') \in [\![P_1*]\!]_{\mathsf{io}} \mid \nexists d'' : (d', d'') \in [\![P]\!]_{\mathsf{io}}\}
\end{aligned}$$

Using control automata, we express the reactive rather than the input-output behaviour of Lang; or in other words, it is a small-step semantics rather than a big-step semantics. An advantage, furthermore, is that we capture the meaning of control expressions without reference to any particular rule system. However, our approach inevitably implies that the meaning of, for instance, **alap** changes with respect to the above definition: rather than considering a sub-expression "possible" if it can run to a successful completion, we consider it "possible" if it can do a *single* step.

The semantics of Lang is defined through a set of operators over CAut corresponding to the constructs of the language. For this purpose, we first need to define what it means for an automaton to *fail*; this is an important concept in the definition of **alap** and **try-else**. The failure of an automaton is based on the non-applicability of its initial actions, being those rules that are scheduled as a first action. This set of initial actions is defined as follows:

Definition 6 (initial control actions). *The initial actions of a given control automaton $\mathcal{C}$ are defined by*

$$Init(\mathcal{C}) = \{n \mid q_0 \xrightarrow{F_i \ldots F_n} \xrightarrow{n}\} \cup \{\delta \mid \exists q' : q_0 \xrightarrow{F_0 \ldots F_n} q' \in S\}$$

A δ in the result indicates that in the given automaton there is a path from the start state to a success state consisting of only failure transitions. The automaton is considered to be always successful. We will see that a failure transition for $\mathcal{C}$ is typically only created when $\delta \notin Init(\mathcal{C})$.

In de definitions below, we use control automata $\mathcal{C}_i = (Q_i, \Sigma, \to_i, q_{0_i}, S_i)$ for $i = 1, 2$, and we use distinct fresh states $q_N, q_M \notin Q_1$. Due to the lack of space

in this paper, we restrict ourselves to the most interesting language constructs: single rules, sequential composition, **alap** and **try**–**else**.

$$\begin{aligned}
\mathcal{C}_{\mathsf{rule}} &= (\{q_M, q_N\}, \Sigma, \{(q_N, \mathsf{rule}, q_M)\}, q_M, \{q_N\}) \\
\mathcal{C}_1;\mathcal{C}_2 &= (Q_1 \setminus S_1 \cup Q_2, \Sigma, \rightarrow, q_{0,1}, S_2), \text{where} \\
&\quad \rightarrow = \rightarrow_1 \setminus \{(q, x, q') \mid q \xrightarrow{x}_1 q' \in S_1\} \cup \{(q, x, q_{0,2}) \mid q \xrightarrow{x}_1 q' \in S_1\} \\
&\qquad \cup \rightarrow_2 \\
\mathcal{C}_1{\downarrow} &= (Q_1 \setminus S_1 \cup \{q_N\}, \Sigma, \rightarrow, q_{0,1}, \{q_N\}), \text{where} \\
&\quad \rightarrow = \rightarrow_1 \setminus \{(q, x, q') \mid q \xrightarrow{x}_1 q' \in S_1\} \cup \{(q, x, q_{0,1}) \mid q \xrightarrow{x}_1 q' \in S_1\} \\
&\qquad \cup \{(q_{0,1}, \mathit{Init}(C_1), q_N) \mid \delta \notin \mathit{Init}(C_1)\} \\
\mathcal{C}_1 \mathbin{?} \mathcal{C}_2 &= (Q_1 \cup Q_2, \Sigma, \rightarrow, q_{0,1}, S_1 \cup S_2), \text{where} \\
&\quad \rightarrow = \rightarrow_1 \cup \rightarrow_2 \cup \{(q_{0,1}, \mathit{Init}(C_1), q_{0,2}) \mid \delta \notin \mathit{Init}(C_1)\}
\end{aligned}$$

The following points are noteworthy:

- In the sequential composition $\mathcal{C}_1;\mathcal{C}_2$, every transition in $\mathcal{C}_1$ to a success state is redirected to the start state of $\mathcal{C}_2$.
- In the "alap closure" $\mathcal{C}_1{\downarrow}$, transitions to success states are redirected to the start state. Optionally, a failure transition is created to a fresh state q_N.
- In the "try-else" operation $\mathcal{C}_1?\mathcal{C}_2$, the start state of $\mathcal{C}_1$ is optionally connected to the start state of $\mathcal{C}_2$ by a failure transition.

Without proof, we state the following property:

Proposition 1. CAut *is closed under the constructions defined above.*

This gives rise to the following semantics function $[\![-]\!]_{\mathsf{aut}} : \mathsf{Lang} \rightarrow \mathsf{CAut}$:

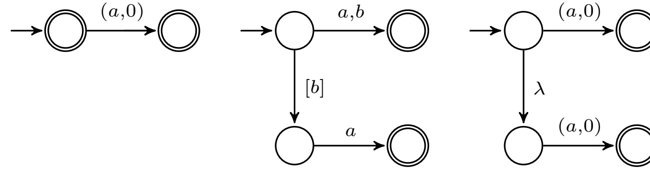
$$\begin{aligned}
[\![\mathsf{rule}]\!]_{\mathsf{aut}} &= \mathcal{C}_{\mathsf{rule}} \\
[\![P_1; P_2]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}; [\![P_2]\!]_{\mathsf{aut}} \\
[\![\textbf{alap } P_1]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}}{\downarrow} \\
[\![\textbf{try } P_1 \textbf{ else } P_2]\!]_{\mathsf{aut}} &= [\![P_1]\!]_{\mathsf{aut}} \mathbin{?} [\![P_2]\!]_{\mathsf{aut}}
\end{aligned}$$

As we have pointed out above, this differs from the semantics studied in [7] (and elsewhere) in the treatment of failure, which for them is the failure of a small step but for them the failure of a big step. For instance, the control program $P = \textbf{alap}\ (\mathsf{a};\mathsf{b});\mathsf{c}$ imposed on a rule system where $d \xrightarrow{\mathsf{a},i} d' \xrightarrow{\mathsf{c},j} d''$ but $\nexists k : d' \xrightarrow{\mathsf{b},k}$ gives rise to $(d, d'') \in [\![P]\!]_{\mathsf{io}}$, whereas $[\![P]\!]_{\mathsf{aut}} \times \mathcal{A}_d$ only contains the transition $(d, q_0) \xrightarrow{\mathsf{a},i} (d', q_1) \notin S$, without further outgoing transitions.

4 Guarded Control Automata

The product operation defined in Definition 5 can result in a system automaton that is non-deterministic in the sense of Def. 2, even when system and control automaton are both deterministic.

Example 3. Consider the following automata:



The system automaton on the left and the control automaton in the middle are both clearly deterministic, in the sense of Def. 2. In their product, shown on the right, the rule application $(a, 0)$ occurs twice, hence this is non-deterministic.

The desired behaviour of the product of a deterministic control automaton and a control automaton is a deterministic controlled system automaton. For that purpose, we introduce *guarded control automata*. Here, every transition consists of a rule name with a positive and negative guard, both of which are sets of rules. For a transition to be enabled, all rules in the negative guard must fail to be applied, and the rules in the positive guard set must be applicable (i.e. the rule must have at least one match). We also introduce a *determinisation* operation for normal control automata that produces a guarded control automaton.

We use the notation $q \xrightarrow{[F|A]n} q'$ for transitions with guards, where F is the positive and A is the negative guard. When $A = \emptyset$, we use the notation $q \xrightarrow{[F]n} q'$; q$\xrightarrow{n}$q' denotes that $F = A = \emptyset$.

Definition 7 (guarded control automaton). *A guarded control automaton is an automaton with* $\Sigma = \mathtt{Fail} \times 2^{\mathsf{Rule}} \times \mathsf{Rule}$ *and* $S \subseteq Q \times \mathtt{Fail}$*. 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* $\exists q \xrightarrow{[F \cup A|\,]n}$.

The class of guarded control automata is denoted GAut. Note that the success states are now also conditional (or guarded).

We define the failure dependency function fd, that returns the union of all possible failures that lead to a state where n is allowed. Whether or not these rules fail determines the precise target control states the system can be in.

Definition 8 (failure dependency). *The failure dependency in a given set of states for a given rule is defined by:*

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

Determinisation of a control automaton is given by a function det:

Definition 9 (control automaton determinisation). *Given a control automaton $\mathcal{C}$, $det(\mathcal{C})$ is an automaton with $Q = 2^{Q_\mathcal{C}} \setminus \emptyset$, $q_0 = \{q_{0,\mathcal{C}}\}$, and $\rightarrow$,S are defined by:*

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

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

The states in the guarded control automaton are sets of states of the original control automaton. The target state of a transition $q \xrightarrow{[A|F]n} q'$ in the guarded control automaton is defined as the set of all states in the original control automaton that can be reached with the failures in F followed by a rule name n. The positive guard A of a transition contains the names of those rules that must be enabled, which are those rules that are not in F but are in the failure dependency of n in the source state. A success state is set of tuples consisting of states and failures. Success is a conditional property; we show in the product definition that a product state is a success state if a failure is satisfied by the system component in the product state. A state with the empty failure is an unconditional success state.

We state the following property:

Proposition 2. *Given a control automaton $\mathcal{C}$, the automaton $det(\mathcal{C})$ is a guarded control automaton.*

The proof can be found in Appendix A.

Example 4. Figure 5 shows the guarded control automaton for the control automaton of Figure 3. From the start state, an e transition is possible to $\{c_0, c_1\}$, which can be repeated as long as e is possible. From both $\{c_0\}$ and $\{c_0, c_1\}$ a c transition is possible to $\{c_2\}$ if e fails. The outgoing transition in $\{c_2\}$ represents the success condition. Since there is only the empty failure set, it is an unconditional success.

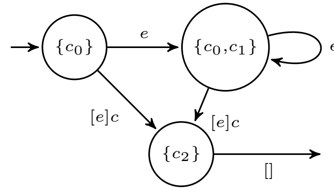


Fig. 5. Example Extended Control Automaton

For the definition of the product of system automata and guarded control automata, we introduce the function *enabled*, which returns a set of rules that are enabled in a given system state or a state reachable after an arbitrary sequence of λ's.

Definition 10 (enabled rules). *Given a system automaton $\mathcal{A}$, the function enabled* $: Q_{\mathcal{A}} \to 2^{\mathsf{Rule}}$ *is defined as:*

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

The semantics of a guarded control automaton is given by the product with a system automaton. This results in another system automaton, where states are tuples of system states and guarded control states, defined as follows:

Definition 11 (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 $\rightarrow$, 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 $\mathcal{G}$ is paired with rule applications of n in the system automaton $\mathcal{A}$ when none of the rules in the negative guard F are applicable in $q_{\mathcal{A}}$, and all rules in the positive guard A are applicable in $q_{\mathcal{A}}$.

The success states are those states where $q_{\mathcal{A}}$ is a success state, and a failure its combined with in S is satisfied in $q_{\mathcal{A}}$.

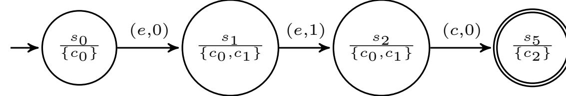


Fig. 6. Example Guarded Product Automaton

Example 5. Figure 6 shows the product of the guarded control automaton of Figure 5 and the system automaton of Figure 1.

As said, the purpose of guarded control automata is to be able to produce deterministic controlled system automata. We state the following property, the proof of which is given in Appendix A:

Proposition 3. *Given a guarded control automaton $\mathcal{G}$ and a deterministic system automaton $\mathcal{A}$, $\mathcal{A} \times \mathcal{G}$ is deterministic.*

4.1 Equivalence

We will now show that guarded control automata serve their intended purpose, namely that the product of a system automaton with a control automaton is essentially the same as its product with the determinised guarded control automaton. "Essentially the same" means that they have the same language in terms of $(\mathsf{Rule} \times Id)$-traces. This is depicted in Figure 7 and Theorem 1.

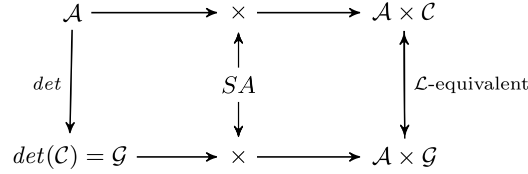


Fig. 7. Schematic Representation of the Equivalence Relationship

Theorem 1. *For all system automata $\mathcal{A}$ and control automata $\mathcal{C}$, $\mathcal{L}(\mathcal{A} \times \mathcal{C}) = \mathcal{L}(\mathcal{A} \times det(\mathcal{C}))$*

To prove this, we show inclusions in both directions, using two distinct notions of simulation. Proof of the claimed properties is provided in Appendix A.

Definition 12 (forward simulation). *Given two system automata $\mathcal{A}_1, \mathcal{A}_2$, a relationship $\rho \subseteq Q_1 \times Q_2$ is called a* forward simulation *if:*

$$(q_{0_1}, q_{0_2}) \in \rho \tag{1}$$

and for all: $(q_1, q_2) \in \rho$:

$$q_1 \xrightarrow{(n,i)} q_1' \Rightarrow \exists q_2 \xrightarrow{(n,i)} q_2' \wedge (q_1', q_2') \in \rho \tag{2}$$

$$\exists q_1 \xRightarrow{\epsilon} q_1' \in S_1 \Rightarrow q_2 \xRightarrow{\epsilon} q_2' \in S_2 \tag{3}$$

Proposition 4. *If there exists a forward simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{T}(\mathcal{A}_1) \subseteq \mathcal{T}(\mathcal{A}_2)$ and $\mathcal{T}^{\surd}(\mathcal{A}_1) \subseteq \mathcal{T}^{\surd}(\mathcal{A}_2)$.*

Proposition 5. *Let $\mathcal{A}$ be a system automaton and $\mathcal{C}$ a control automaton; then the relation defined by $\rho = \{((q_\mathcal{A}, q_\mathcal{C}), (q_\mathcal{A}, qs)) \mid q_\mathcal{C} \in qs\}$ is a forward simulation between $\mathcal{A} \times \mathcal{C}$ and $\mathcal{A} \times det(\mathcal{C})$.*

The other direction is also covered by a simulation.

Definition 13 (reverse simulation). *Given two system automata $\mathcal{A}_1$ and $\mathcal{A}_2$, $\rho \subseteq Q_2 \times 2^{Q_1}$ is called a* reverse simulation *if:*

$$(q_{0,2}, \{q_{0,1}\}) \in \rho \tag{4}$$

and for all $(q_\mathcal{A}, R) \in \rho$:

$$q_2 \xrightarrow{(n,i)}_2 q_2' \Rightarrow \exists (q_2', R') \in \rho . \forall r' \in R' . \exists r \in R . r \xRightarrow{(n,i)}_1 r' \tag{5}$$

$$q_2 \in S_2 \Rightarrow \exists q_1 \in R . q_1 \xRightarrow{\epsilon} q_1' \in S_1 \tag{6}$$

Proposition 6. *If there exists a reverse simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{T}(\mathcal{A}_2) \subseteq \mathcal{T}(\mathcal{A}_1)$ and $\mathcal{T}^{\surd}(\mathcal{A}_2) \subseteq \mathcal{T}^{\surd}(\mathcal{A}_1)$.*

Proposition 7. *Let $\mathcal{A}$ be a system automaton and $\mathcal{C}$ a control automaton, and $\mathcal{G} = det(\mathcal{C})$, then the relation defined by $\rho = \{((q_\mathcal{A}, q_\mathcal{G}), R) \mid \forall q_\mathcal{C} \in q_\mathcal{G} : (q_\mathcal{A}, q_\mathcal{C}) \in R\}$ is a reverse simulation between $\mathcal{A} \times det(\mathcal{C})$ and $\mathcal{A} \times \mathcal{C}$.*

With the help of the above results, we can now prove the theorem.

Proof (Theorem 1). From propositions 4 and 5 it follows that $\mathcal{T}(\mathcal{A}\times\mathcal{C}) \subseteq \mathcal{T}(\mathcal{A}\times det(\mathcal{C}))$ and $\mathcal{T}^{\checkmark}(\mathcal{A}\times\mathcal{C}) \subseteq \mathcal{T}^{\checkmark}(\mathcal{A}\times det(\mathcal{C}))$; from propositions 6 and 7 it follows that the inverse inclusions also hold. This implies the proof obligation.

5 Conclusions

5.1 Contributions

In this paper, we have presented an automaton formalism for controlling rule applications in rule-based systems. We have introduced the notion of failure as the non-applicability of a set of rules in a certain state, and use these failures failures as an element of control in our automata. The resulting behaviour is defined as the product with a system automaton, an automaton representation of the uncontrolled rule system. The result is a reactive semantics for control expressions.

We have introduced a formalism for guarded control automata and the corresponding product operation, that — when applied to a deterministic system automaton — results in a deterministic controlled system automaton. We have proved that the languages of products using a normal and corresponding guarded control automaton coincide.

We have implemented (a superset of) the language presented here in the tool GROOVE [15].

In future work, we like to extend the described control expressions with features such as atomic procedures, transactions and rule parameters.

5.2 Related Work

There are two important areas of related work: on the one hand, other approaches to add control to rule-based systems, and on the other, results from process algebra.

Other approaches to control. Although we have presented this work in the general context of rule-based systems, as far as we are aware most of the work on explicit control for such systems has been done for the special case of *graph transformation systems*; therefore, this is what we will focus on.

First let us remark that in this paper we have concentrated on one particular method of controlling rule applicability. There are others, such as application conditions (e.g., [5, 6]), which have their own advantages and may very well be used in conjunction with control expressions.

One of the first developments in the direction of using explicit control expressions for graph transformation was the PROGRES environment; see, e.g., [16, 18]. The aim here was to obtain a powerful and usable framework for programmed graph transformation, rather than study the theoretical properties of such a framework (although a translation into flow graphs was studied in [22]).

In the same vein, the control languages in tools like VIATRA2 [20] and VMTS [12] stress power and usability over theoretical properties. FUJaBA has the appealing graphical *storyboard* language for control [4], but here the control is completely integrated with the rules and does not lend itself at all to an analysis like the one in this paper.

The theory behind control conditions for rule-based systems has been studied intensively in the context of *graph transformation units* by Kreowski, Kuske and others; see, for instance, [9–11]. However, they take a perspective that is quite different from ours, and indeed represents the input-output interpretation discussed in Section 3, insisting that "every description of a binary relation on graphs may be used as a control condition". Also in that interpretation, Schurr [17] contains a systematic discussion of various operators and their meaning, and Habel and Plump in [7] show the minimality of the control language consisting only of choice, sequential composition and **alap**. In [14], a **while–do** is added to this language, to increase usability. We strongly believe that also the **try–else** operator of this paper is useful in that regard.

Finally, it is worth mentioning that the tools GrEAT [21] and ATOM^3 [19] include facilities for rule scheduling, but based on *data flow* rather than control flow.

Failures in process algebra. A weak link exists to the concept of failure semantics in process algebra, as developed in [1], leading to the formalism of CSP [8]; and a slightly stronger one to the refusal testing semantics of [13], where failures can be interspersed with ordinary actions. The failures in those papers, however, are solely used as *observations* of the execution capabilities of a process, never to *control* the process. Thus, despite the superficial similarities, the models and their purpose are quite different from the research reported in this paper.

5.3 Scalability & Implementation

An unfortunate consequence of guarded control automata is the fact that there is an exponential blow-up in the possible number of states ($|Q_{\mathcal{G}}| = 2^{|Q_C|}$) *and* transitions in applying the *det* function ($|{\rightarrow_{\mathcal{G}}}|$ is in the order of $2^{|\rightarrow_C|}$) due to the adding of positive and negative guards.

In practice only a small portion of these states will actually be reachable. To deal with this problem, the implementation of this work in [15] uses a lazy construction of $\mathcal{G}$ states during generation of the guarded system automaton. In this way, only those states are created that are (obviously) possible, but - more important - are in fact *used.* In a small test case that we have performed using a control program and a corresponding control automaton with 19 states and 9 failure transitions, only 17 distinct guarded control states were used during generation of the guarded system automaton (of a maximum possible count of 524288). In our experience, a single Rule is rarely used twice in a control program, limiting the number of guarded control states and the number of transitions.

References

1. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM **31**(3) (1984) 560–599
2. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Theory and Application of Graph Transformations (TAGT). In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Theory and Application of Graph Transformations (TAGT). Volume 1764 of LNCS., Springer (2000)
3. Ehrig, H., Engels, G., Rozenberg, G., Kreowski, H.J., eds.: Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Theory. Volume 2. World Scientific (1999)
4. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the unified modeling language and Java. [2] 296–309
5. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3/4) (1996) 287–313
6. Habel, A., Pennemann, K.H.: Nested constraints and application conditions for high-level structures. In Kreowski, H.J., et al., eds.: Formal Methods in Software and Systems Modeling. Volume 3393 of LNCS., Springer (2005) 293–308
7. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In: Foundations of Software Science and Computation Structures (FoSSaCS). Volume 2030 of LNCS., Springer (2001) 230–245
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
9. Kreowski, H.J., Kuske, S.: On the interleaving sematics of transformation units - a step into GRACE. In Cuny, J.E., Ehrig, H., Engels, G., Rozenberg, G., eds.: Graph Gramars and Their Application to Computer Science (TAGT). Volume 1073 of LNCS., Springer (1996) 89–106
10. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. [3] 607–638
11. Kuske, S.: More about control conditions for transformation units. [2] 323–337
12. Lengyel, L., Levendovszky, T., Mezei, G., Charaf, H.: Model Transformation with a Visual Control Flow Language. International Journal of Computer Science (IJCS) **1**(1) (2006) 45–53
13. Phillips, I.: Refusal testing. In Kott, L., ed.: Automata, Languages and Programming (ICALP). Volume 226 of LNCS., Springer (1986) 304–313
14. Plump, D., Steinert, R.: Towards Graph Programs for Graph Algorithms. In: In Proc. International Conference on Graph Transformation (ICGT 2004, Springer (2004) 128–143
15. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz, J.L., Nagl, M., Böhlen, B., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE). Volume 3062 of LNCS., Berlin, Springer Verlag (2004) 479–485
16. Schürr, A.: Introduction to PROGRES, an attribute graph grammar based specification language. In Nagl, M., ed.: Graph-Theoretic Concepts in Computer Science. Volume 411 of LNCS., Springer (1990) 151–165
17. Schurr, A.: Programmed graph replacement systems. In Rozenberg, G., ed.: Handbook on Graph Grammars: Foundations. Volume 1., World Scientific (1997) 479–546
18. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. [3] 487–550
19. Syriani, E., Vangheluwe, H.: Programmed graph rewriting with DEVS. In Nagl, M., Schürr, A., eds.: Applications of Graph Transformations with Industrial Relevance (AGTIVE). LNCS, Springer (2007)

20. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. **68**(3) (2007) 214–234
21. Vizhanyo, A., Neema, S., Shi, F., Balasubramanian, D., Karsai, G.: Improving the usability of a graph transformation language. ENTCS **152** (2006) 207–222
22. Zündorf, A., Schürr, A.: Nondeterministic control structures for graph rewriting systems. In Schmidt, G., Berghammer, R., eds.: Graph-Theoretic Concepts in Computer Science. Volume 570 of LNCS., Springer (1992) 48–62

A Proofs

This appendix contains proofs of all the results of this paper.

Proposition 2 Given a control automaton $\mathcal{C}$, the automaton $det(\mathcal{C})$ is a guarded control automaton.

Proof. The proof that the created automaton satisfies the requirements (1), (2), and (3) of Def. 7 follows directly from the construction of $\rightarrow$ in Def. 9. Let $\mathcal{G} = det(\mathcal{C})$. For all $q_\mathcal{G} \in Q_\mathcal{G}$ with $q_\mathcal{G} \xrightarrow{[F_1|A_1]n}$ and $q_\mathcal{G} \xrightarrow{[F_2|A_2]n}$ it follows that:

1. $FD = fd(q_\mathcal{G}, n), F_i \subseteq FD, A_i = FD \setminus F$, which implies that $A \cap F = \emptyset$.
2. $FD = fd(q_\mathcal{G}, n)$ and $A_1 = F_2 \cup A_2 = FD$.
3. $q_\mathcal{G} \xrightarrow{[F|\,]n} q'_\mathcal{G}$ is the transition where $F = fd(q, n), A = \emptyset$. Given Def. 8, $q'_\mathcal{G}$ exists.

Proposition 3 Given a guarded control automaton $\mathcal{G}$ and a deterministic system automaton $\mathcal{A}$, $\mathcal{A} \times \mathcal{G}$ is deterministic.

Proof. Given that $\mathcal{A}$ is deterministic, it contains no λ-transitions. Then, by construction (Def. 11) P contains no λ-transitions either. Let $P = \mathcal{A} \times \mathcal{G}$. For all $(q_\mathcal{A}, q_\mathcal{G}) \in Q_P$ with $(q_\mathcal{A}, q_\mathcal{G}) \xrightarrow{(n,i)} (q'_\mathcal{A}, q'_\mathcal{G})$ and $(q_\mathcal{A}, q_\mathcal{G}) \xrightarrow{(n,i} (q''_\mathcal{A}, q''_\mathcal{G})$ we need to show that $(q'_\mathcal{A}, q'_\mathcal{G}) = (q''_\mathcal{A}, q''_\mathcal{G})$. Since $\mathcal{A}$ is deterministic, we know that $q_\mathcal{A} \xrightarrow{(n,i)}_\mathcal{A} q'_\mathcal{A}$ and $q_\mathcal{A} \xrightarrow{(n,i)}_\mathcal{A} q''_\mathcal{A}$ implies that $q'_\mathcal{A} = q''_\mathcal{A}$. Also, for $q_\mathcal{G} \xrightarrow{[F_1|F_2]n}_\mathcal{G} q'_\mathcal{G}$ and $q_\mathcal{G} \xrightarrow{[F_1|F_2]n}_\mathcal{G} q''_\mathcal{G}$, Def. 7 implies that $q'_\mathcal{G} = q''_\mathcal{G}$. For $q_\mathcal{G} \xrightarrow{[F|A]n}_\mathcal{G}$, there is at most one transition where $F \cap enabled(q_\mathcal{A}) = \emptyset$ and $A \subseteq enabled(q_\mathcal{A})$. Def. 11 implies that $(q'_\mathcal{A}, q'_\mathcal{G}) = (q''_\mathcal{A}, q''_\mathcal{G})$.

Proposition 4 If there exists a forward simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{T}(\mathcal{A}_1) \subseteq \mathcal{T}(\mathcal{A}_2)$ and $\mathcal{T}^\checkmark(\mathcal{A}_1) \subseteq \mathcal{T}^\checkmark(\mathcal{A}_2)$.

Proof. Following the definition of the language of an automaton as the set of all traces, we must show that $\mathcal{T}(\mathcal{A}_1) \subseteq \mathcal{T}(\mathcal{A}_2)$ and $\mathcal{T}^\checkmark(\mathcal{A}_1) \subseteq \mathcal{T}^\checkmark(\mathcal{A}_2)$. Proof is given by induction over the length of the traces.

Hypothesis ($\mathcal{T}$) For all traces $q_{0,1} \stackrel{w}{\Longrightarrow} q'_1$ with $|w| = n$, there is a $q_{0,2} \stackrel{w}{\Longrightarrow} q'_2$ with $q'_1 \rho q'_2$.

Basis. The trace w with $|w| = 0$ ends in $q_{0,1}$ and $q_{0,2}$, and $q_{0,1} \rho q_{0,2}$.

Inductive Step. Let the hypothesis hold for all traces w with $|w| = n$. For all traces w', with $|w'| = n + 1$, there exists a trace $w, |w| = n$, such that $w' = w; (n, i)$. Also, there is a $q_1 \stackrel{(n,i)}{\Longrightarrow} q'_1$. From Def. 12(2) it follows that there is also a transition $q'_2 \stackrel{(n,i)}{\Longrightarrow} q''_2$, and that $q''_1 \rho q''_2$.

We must do the same for the successful traces. Since $\mathcal{T}^\checkmark(\mathcal{A}_1) \subseteq \mathcal{T}(\mathcal{A}_2)$, we can take a shortcut here, by referring to the hypothesis above for all traces, which we already have proved to be true. For all traces $w \in \mathcal{T}^\checkmark(\mathcal{A}_1)$, if follows that $w \in \mathcal{T}(\mathcal{A}_1)$. Then there is a $(q_1, q_2) \in \rho$ with $q_{0,1} \stackrel{w}{\Longrightarrow} q_1$ and $q_{0,2} \stackrel{w}{\Longrightarrow} q_2$. Also, there exist a $q_1 \stackrel{\epsilon}{\Longrightarrow} q'_1 \in S_1$. From Def. 12(3) it follows that there is a $q_2 \stackrel{\epsilon}{\Longrightarrow} q'_2 \in S_2$, thus $w \in \mathcal{T}^\checkmark(\mathcal{A}_2)$.

Proposition 5 Let $\mathcal{A}$ be a system automaton and $\mathcal{C}$ a control automaton; then the relation defined by $\rho = \{((q_\mathcal{A}, q_\mathcal{C}), (q_\mathcal{A}, qs)) \mid q_\mathcal{C} \in qs\}$ is a forward simulation between $\mathcal{A} \times \mathcal{C}$ and $\mathcal{A} \times det(\mathcal{C})$.

Proof. We now give the proofs that (1), (2) and (3) of Def. 12 hold for the proposed ρ. Let $\mathcal{A}_1 = \mathcal{A} \times \mathcal{C}, \mathcal{G} = det(\mathcal{C}), \mathcal{A}_2 = \mathcal{A} \times \mathcal{G}$.

(1) By construction (Def. 5 and Def. 11) $q_{0,1} = (q_{0,\mathcal{A}}, q_{0,\mathcal{C}})$ and $q_{0,2} = (q_{0,\mathcal{A}}, \{q_{0,\mathcal{C}}\})$; hence it follows that $q_{0,1} \rho q_{0,2}$.

(2) Let $(q_\mathcal{A}, q_\mathcal{C})\rho(q_\mathcal{A}, q_\mathcal{G})$ and let there be a transition $(q_\mathcal{A}, q_\mathcal{C}) \xRightarrow{(n,i)}_1 (q'_\mathcal{A}, q'_\mathcal{C})$. Def. 5 implies $q_\mathcal{C} \xrightarrow{F_1 \ldots F_n} \xrightarrow{n} q'_\mathcal{C}$, $F \cap enabled(q_\mathcal{A}) = \emptyset$, with $F = F_1 \cup \ldots \cup F_n$, and $q_\mathcal{A} \xRightarrow{\epsilon} \xrightarrow{(n,i)} q'_\mathcal{A}$. Let $F' = fd(q_\mathcal{G}, n) \setminus enabled(q_\mathcal{A})$. Since $q_\mathcal{C} \in q_\mathcal{G}$, Def. 7 implies $F \subseteq F'$, $q_\mathcal{G} \xrightarrow{[A|F]n)} q'_\mathcal{G}$ with $A = F' \setminus F$, and $q'_\mathcal{C} \in q'_\mathcal{G}$. From Def. 11 it follows that $(q_\mathcal{A}, q_\mathcal{G}) \xRightarrow{\epsilon} \xrightarrow{(n,i)}_2 (q'_\mathcal{A}, q'_\mathcal{G})$, thus $(q'_\mathcal{A}, q'_\mathcal{C})\rho(q'_\mathcal{A}, q'_\mathcal{G})$.

(3) Let $(q_\mathcal{A}, q_\mathcal{C})\rho(q_\mathcal{A}, q_\mathcal{G})$, and $\exists (q_\mathcal{A}, q_\mathcal{C}) \xRightarrow{\epsilon}_1 (q'_\mathcal{A}, q'_\mathcal{C}) \in S_1$. Def. 5 implies $q_\mathcal{A} \in S_\mathcal{A}$, $\exists q_\mathcal{C} \xrightarrow{F_1 \ldots F_n} q'_\mathcal{C} \in S_\mathcal{C}, F = F_1 \cup \ldots \cup F_n$, and $F \cap enabled(s) = \emptyset$. Def. 9 implies $(q_\mathcal{G}, F) \in S_\mathcal{G}$. Finally, from Def. 11 follows $(q_\mathcal{A}, q_\mathcal{G}) \in S_2$.

Proposition 6 If there exists a reverse simulation between $\mathcal{A}_1$ and $\mathcal{A}_2$, then $\mathcal{T}(\mathcal{A}_2) \subseteq \mathcal{T}(\mathcal{A}_1)$ and $\mathcal{T}^\surd(\mathcal{A}_2) \subseteq \mathcal{T}^\surd(\mathcal{A}_1)$.

Proof. Following the definition of the language of an automaton as the set of all traces, we must show that $\mathcal{T}(\mathcal{A}_2) \subseteq \mathcal{T}(\mathcal{A}_1)$ and $\mathcal{T}^\surd(\mathcal{A}_2) \subseteq \mathcal{T}^\surd(\mathcal{A}_1)$. Proof is given by induction over the length of the traces.

Hypothesis ($\mathcal{T}$). For all traces $q_{0,2} \xRightarrow{w} q'_2$ with $|w| = n$, there is a R', such that for all $q'_1 \in R'$, there is a $q_{0,1} \xRightarrow{w} q'_1$ and $q'_2 \rho R'$.

Basis. The trace w with $|w| = 0$ ends in $q_{0,1}$ and $q_{0,2}$, and $q_{0,2}\rho\{q_{0,1}\}$.

Inductive Step. Let the hypothesis hold for all traces w with $|w| = n$. For all traces w' with $|w'| = n+1$, there is a w such that $w' = w; (n,i)$. Then there is a $(q_2, R) \in \rho$, with trace w ending in q_2 and all $q_1 \in R$. Also, there is a transition $q_2 \xRightarrow{(n,i)} q'_2$. From Def. 13(5) it follows that there is an R', such that for all $q'_1 \in R'$, there is a q_1 in R with $q_1 \xRightarrow{(n,i)} q'_1$ with $q'_2 \rho R'$.

For the successful traces we again take a shortcut by using the proof for all traces. Let $w \in \mathcal{T}^\surd(\mathcal{A}_2)$, then also $w \in \mathcal{T}(\mathcal{A}_2)$. Then there is a $(q_2, R) \in \rho$ and trace w to q_2 and to all $q_1 \in R$. Since $w \in \mathcal{T}^\surd(\mathcal{A}_2)$, there is a $q_2 \xRightarrow{\epsilon} q'_2 \in S_2$. From Def. 13(6) it follows that there is a $q_1 \in R$ for which there is a $q_1 \xRightarrow{\epsilon} \in S_1$. Thus, $w \in \mathcal{T}^\surd(\mathcal{A}_1)$.

Proposition 7 Let $\mathcal{A}$ be a system automaton and $\mathcal{C}$ a control automaton, and $\mathcal{G} = det(\mathcal{C})$, then the relation defined by $\rho = \{((q_\mathcal{A}, q_\mathcal{G}), R) \mid \forall q_\mathcal{C} \in q_\mathcal{G} : (q_\mathcal{A}, q_\mathcal{C}) \in R\}$ is a reverse simulation between $\mathcal{A} \times det(\mathcal{C})$ and $\mathcal{A} \times \mathcal{C}$.

Proof. We now give the proofs that (4), (5) and (6) of Def. 13 hold for the proposed ρ. Let $\mathcal{A}_1 = \mathcal{A} \times \mathcal{C}, \mathcal{A}_2 = \mathcal{A} \times det(\mathcal{C}), \mathcal{G} = det(\mathcal{C})$.

(4) By construction (Def. 5 and 11) $q_{0,1} = (\mathcal{A}, \mathcal{C})$ and $q_{0,2} = (\mathcal{A}, \{\mathcal{C}\})$; hence it follows that $q_{0,2}\rho\{q_{0,1}\}$.

(5) Let $(q_{\mathcal{A}}, q_{\mathcal{G}})\rho R$ and let there be a transition $(q_{\mathcal{A}}, q_{\mathcal{G}} \xrightarrow{(n,i)}_2 (q'_{\mathcal{A}}, q'_{\mathcal{G}})$. From Def. 11 it follows there is a $q_{\mathcal{A}} \overset{\epsilon}{\Longrightarrow} \xRightarrow{(n,i)}_{\mathcal{A}} q'_{\mathcal{A}}$ and $q_{\mathcal{G}} \xrightarrow{([A|F]n)} q'_{\mathcal{G}}$, such that $F \cap enabled(s) = \emptyset$, and $A \subseteq enabled(s)$. Def. 7 implies $q'_{\mathcal{G}} = \{q'_{\mathcal{C}} \mid \exists q_{\mathcal{C}} \in q_{\mathcal{G}} : q_{\mathcal{C}} \xrightarrow{F_1 \ldots F_n} \xrightarrow{n} q'_{\mathcal{C}}$ with $F_1 \cup \ldots \cup F_n \subseteq F\}$. Def. 5 implies $\forall q'_{\mathcal{C}}.\exists q_{\mathcal{C}} \in q_{\mathcal{G}}.(q_{\mathcal{A}}, q_{\mathcal{C}}) \xRightarrow{(n,i)}_1 (q'_{\mathcal{A}}, q'_{\mathcal{C}}) \in R'$.

(6) Let $(q_{\mathcal{A}}, q_{\mathcal{G}})\rho R$ and $(q_{\mathcal{A}}, q_{\mathcal{G}}) \in S_2$. Def. 11 implies $q_{\mathcal{A}} \in S_{\mathcal{A}}, \exists F.(q_{\mathcal{G}}, F) \in S_{\mathcal{G}} : F \cap enabled(s) = \emptyset$. Def. 7 implies $\exists q_{\mathcal{C}} \in q_{\mathcal{G}} : q_{\mathcal{C}} \xrightarrow{F_1 \ldots F_n}_{\mathcal{C}} q'_{\mathcal{C}} \in S_c, F_1 \cup \ldots \cup F_N = F$. Finally, Def. 5 implies $(q_{\mathcal{A}}, q_{\mathcal{C}}) \in R \wedge (q_{\mathcal{A}}, q_{\mathcal{C}}) \overset{\epsilon}{\Longrightarrow} (q'_{\mathcal{A}}, q'_{\mathcal{C}}) \in S_1$.