Softw Syst Model (2007) 6:381–399
DOI 10.1007/s10270-006-0037-5

REGULAR PAPER

Computing refactorings of state machines

Alexander Pretschner · Wolfgang Prenninger

Received: 3 February 2006 / Revised: 16 May 2006 / Accepted: 8 September 2006 / Published online: 24 January 2007
© Springer-Verlag 2007

Abstract For behavior models expressed in statechart-like formalisms, we show how to compute semantically equivalent yet structurally different models. These refactorings are defined by user-provided logical predicates that partition the system's state space and that characterize coherent parts – modes or control states – of the behavior. We embed the refactorings into an incremental development process that uses a combination of both tables and graphically represented state machines for describing systems.

1 Introduction

The use of explicit models is enjoying an increasing popularity in the development of complex systems. Modeling languages have matured to a point where they are useful for many developers. Consequently, there is a plethora of tools that enable one to specify systems with these languages. Behavior models are then used to generate simulation and production code [5,14,15] or test cases [4,16,37,38]. They are also subjected to formal verification technology such as model checking [8,12] or automated deductive theorem proving [2,3]. While there is no fit to all needs yet, the respective technology is impressive, and systems of considerable complexity can be handled, as witnessed by the systems described in the – here necessarily incompletely – cited literature.

The increasing complexity of these systems necessitates the study of the evolution of the models itself. The context of this paper is the incremental development of models that represent state machines: models described by means of sequence diagrams, activity diagrams, Petri nets, temporal logics, or process algebras are not in the scope. We study one particular development step, namely refactoring [17,29,31]. Refactoring denotes structural transformations of a software system that do not change its externally visible behavior, except maybe for memory allocation or required processor cycles (and we hence exclude hard real-time systems from our treatise). The main goal of refactorings is to improve a certain quality characteristic of the model (or source code), for example by removing redundancy, reducing complexity, and increasing comprehensibility or reusability. Code-based examples for refactorings include the definition of a function or introduction of a common super class to avoid duplicate code.

We consider refactorings of finite state machines with I/O capabilities and access to an extra data state. This is an add-on to the transitions between the control states in finite state machines that are usually depicted as arrows and circles. For each transition, the guard and the assignments to the data space are specified in a well-defined action language. Our work builds on experience with the CASE tool AutoFocus [22] that we used to model industry-size systems to the end of test case generation [37,38, 41]. Building a model reflects the process of understanding the requirements. The use of state machines forces one to define the control states of this machine early in the development. Sometimes this decision turns out to be inadequate, and different or additional control states

Communicated by Dr. Lionel Briand.

A. Pretschner (✉)
Information Security, ETH Zürich,
8092 Zürich, Switzerland
e-mail: Alexander.Pretschner@inf.ethz.ch.

W. Prenninger
BMW Group, 80788 München, Germany

have to be defined. In the worst case, with current tools, the complete state machine has to be redrawn, a tedious and error-prone task.

Control states can be interpreted as names of predicates over the state space. Given a state machine and a set of such predicates, we show how to compute the transitions (arrows) between the corresponding new control states. Consider a state machine that models a stack (Fig. 1 in Sect. 2.3): one control state with at least three looping transitions, *push*, *pop*, and *get*. Given two predicates that specify that the stack is empty (p) or not empty (q), we show how to compute the transitions between p and q (Fig. 3 in Sect. 4.3). Our main motivation for refactorings of the said kind is the insight that the control states of a behavior model were inadequately chosen. A further motivation is the desire for complementary views on the system [25]. We do not discuss how to pick p and q. The approach is prototypically implemented; yet, the seamless integration into a CASE tool is the subject of future work.

We present our ideas on the grounds of the simple example of a stack. As a proof of concept, we show how our techniques were applied in a case study concerned with testing an automotive network controller [38]. We concentrate on one single flat state machine: parallel composition and hierarchical states are not in the scope.

Our work is based on a development process that uses tables (see Table 1 in Sect. 2.3 for an example) like those in the Software Cost Reduction (SCR) approach [20,21,33]. Unless they grow too large, tables are easy to understand, and one of their important advantages is that they are comparably easy to manipulate. Tool support for manipulating and checking consistency or completeness of different flavors of tables has been around for some time [21,34]. On the other hand, tables are not always utterly convincing to customers who sometimes prefer equivalent graphically displayed executable state machines. We also found that converting tables into a different representation, namely that of equivalent state transition diagrams, is a valuable aid in reviewing the models. In sum, we believe that both tables and graphically represented state machines are valuable in the development process of models. This is consistent with the findings of Parnas and his colleagues that there is a need for more than one kind of tables [32,43]. In this paper, we show how several different state transition diagrams can be computed from one table, and we use the same technology to compute refactorings of a given state transition diagram.

To summarize, we tackle the following *problem*. In the context of incremental development, assume a state machine or a table, and a logical characterization of the different parts of the state space, to be given. How can we compute an equivalent state machine with a set of control states characterized by the logical predicates? The *solution* is the formal definition of the transformation and its prototypical implementation. Our *contribution* is, to our knowledge, the first formal treatment of refactorings of state machines on the grounds of partitionings of the state space. Our approach generalizes to other formalisms as well. Statecharts, for instance, can access any data definitions of a UML model. By translating the statechart into the (standard) formalism given in this paper, we can directly apply our approach, provided that only direct assignments and output are allowed in the action part of a transition.

The remainder of this article is organized as follows. Section 2 introduces some formalism and defines the notions of rule systems, state machines, state transition diagrams, and tables. Section 3 considers the development steps in incremental development processes of behavior models, using both tables and state transition diagrams. Given a logical characterization of the state space, Sect. 4 shows how to compute refactorings. Implementation issues are considered in Sect. 5. Section 6 presents the application of the approach in an industrial case study. Sections 7 and 8 present related work and conclude.

2 Modeling constructs

In this section, we define the notion of *rule systems*. Roughly, rule systems are programs in a language of guarded commands [13]. *Tables* are textual representations of rule systems. *State machines* are a special kind of rule systems with *state transition diagrams* as their graphical representation. The usefulness of and need for these different representations will become apparent later. Before precisely formulating our refactoring steps, we have to introduce some formalism. It borrows from Breitling and Philipps [6].

2.1 Preliminaries

Let V denote a finite set of typed variables. A valuation β maps a variable to a term of its type. A_V is the set of all valuations for a set of variables, V. Let *free*(Φ) denote the set of free variables in a logical formula Φ. In case an assertion Φ evaluates to true when all $v \in$ *free*(Φ) are replaced by $\beta(v)$, we write $\beta \models \Phi$.

Variable names also occur in primed form (the intuition is given in the next paragraph 2.2 on rule systems). For instance, if v is a variable, then priming yields a new variable, v'. Natural extensions apply (1) to sets of variables: $V' = \{v' | v \in V\}$, (2) to valuations: for

$\beta \in A_V$, we have $\beta' \in A_{V'}$ with $\beta'(v') = \beta(v)$ for all $v \in V$, and (3) to assertions: if Φ is an assertion, then Φ' is the assertion that results from priming all variables in *free*(Φ). Unprimed valuations assign values to unprimed variables only, and primed valuations assign values to primed variables only. If an assertion Φ contains both primed and unprimed variables, two valuations are needed for evaluations. We write $\beta, \gamma' \models \Phi$ in case Φ evaluates to true when all unprimed variables v in *free*(Φ) are replaced by $\beta(v)$, and all primed variables v' are replaced by $\gamma'(v')$.

Two valuations $\beta, \gamma \in A_V$ coincide on a subset $W \subseteq V$, denoted $\beta \overset{W}{=} \gamma$, if $\forall v \in W \bullet \beta(v) = \gamma(v)$. Extensions naturally apply to sequences of valuations – $\beta_1\beta_2 \ldots \overset{W}{=} \gamma_1\gamma_2 \ldots$ denotes $\beta_k \overset{W}{=} \gamma_k$ for all k – and to sets of sequences: for two sets of sequences of valuations Y_1 and Y_2, $Y_1 \overset{W}{=} Y_2$ is shorthand for $\forall y_1 \in Y_1 \exists y_2 \in Y_2 \bullet y_1 \overset{W}{=} y_2$ and $\forall y_2 \in Y_2 \exists y_1 \in Y_1 \bullet y_2 \overset{W}{=} y_1$.

$\mathcal{T}(\Sigma, X)$ denotes the set of terms over a signature Σ and a set X of variables. We assume a fixed signature to be given. It defines types, names of functions and data constructors in the action language that is used in guards and assignments of transitions in rule systems (Sect. 2.2). The type of a term t is denoted by *type*(t). Two terms are unifiable $(l \cong r)$ iff $\exists \beta \in A_{V_l \cup V_r} \bullet \beta(l) = \beta(r)$, where V_l and V_r are the sets of variables in l and r, respectively, and $V_l \cap V_r = \emptyset$.

Given a predicate p, $p[f_w/w]_{w \in W}$ denotes the replacement of all variables w in W by terms f_w of the same type. $p'[f_w/w']_{w \in W}$ applies the same notion to replacing primed variables. Finally, function composition is denoted by $\circ$, $\forall x \bullet (f \circ g)(x) = f(g(x))$. The identity mapping is called *id*.

2.2 Rule systems

A *rule system* is a tuple $R = (V, S, T)$, consisting of variables V, initial states S, and a transition relation T.

V consists of disjoint sets of typed variables, I, O, L. They denote input, output, and local variables, respectively. I and O form the *interface* of the rule system, and are also called input and output *ports*.

A *state* of R is a valuation $\beta \in A_V$ that type-correctly maps all variables in V to terms that do not contain variables nor function symbols. $\beta \in A_L$ is called a *data state* of R.

S is an assertion with *free*$(S) \subseteq V$. It describes the *initial state(s)*, and we require S to be satisfiable: $\exists \beta \in A_V \bullet \beta \models S$.

T is a set of *transitions*. Each $t \in T$ is an assertion with *free*$(t) \subseteq V \cup V'$. It relates states to successor states. Unprimed variables are evaluated in the current state, and primed variables are evaluated in the successor state.

We require all transitions $t \in T$ to be of the form $in \wedge g \wedge a \wedge out$. *in* and *out* read input values and compute and write output values, respectively. g is a guard; it defines conditions on the input and the current values of the variables in L. a assigns new values to the variables in L.

More precisely, *in* is a statement of the form $\bigwedge_{i \in I} i \cong \pi_i$ where π_i is a pattern that may contain free *transition-local* variables, H_t, with $H_t \cap V = \emptyset$. We assume $\pi_i \in \mathcal{T}(\Sigma, H_t)$ and *type*(π_i) = *type*(i). The idea is that these variables are bound at runtime, and the values can be used in the computation of guards, output values, and assignments. We naturally extend the notions of states by stipulating that states be elements of $A_{V \cup H_R}$ where $H_R = \bigcup_{t \in T} H_t$. The guard g is a conjunction of predicates over $\mathcal{T}(\Sigma, H_t \cup L)$ with *type*(g) = *Bool*. The assignment $a \equiv \bigwedge_{l \in L} l' = f_l$ type-correctly assigns values to the variables in L', and it may do so by referring to the variables in $L \cup H_t$: $f_l \in \mathcal{T}(\Sigma, L \cup H_t)$ with *type*(f_l) = *type*(l). Finally, $out \equiv \bigwedge_{o \in O} o' = f_o$ assigns values to the output variables, O'. It may refer to the variables in $L \cup H_t$: $f_o \in \mathcal{T}(\Sigma, L \cup H_t)$ with *type*(f_o) = *type*(o).

ε denotes the absence of signals both for input and output ports; types are lifted correspondingly.

Without loss of generality, we will assume that the action language for guards and assignments is a simple first-order functional language without explicit quantifiers, i.e., all variables are free. The reason for this choice is that this is the language supported by the CASE tool AutoFocus which was used in our studies.

A *trace* of a rule system is an infinite sequence of states, $\beta_1\beta_2, \ldots$, with $\beta_i \in A_{V \cup H_R}$. The set of all traces, i.e., the semantics of a rule system, R, is denoted by $[\![R]\!]$. We require $\beta_1 \models S$ and $\forall o \in O \bullet \beta_1(o) = \varepsilon$ – output can only be produced after or during the first transition. Subsequent valuations of a trace, β_n and β_{n+1}, are related by a transition in T: $\forall n \bullet \beta_n, \beta'_{n+1} \models \bigvee_{t \in T} t$. Clearly, there is room for many classical constraints such as causality [7], input enabledness [26], fairness, etc. Rule systems need not be total nor deterministic, and consequently, the models that we consider need not be either.

2.3 State machines, tables, and state transition diagrams

A *state machine* is a rule system with a dedicated variable *state* of a finite type. It specifies the *control state* or *mode* of the state machine. We require an initial control state to be determined in the initial assertion S, each guard to contain a statement $state = \overline{src}$, and each assignment to contain a statement $state' = \overline{dst}$ where $\overline{src}$ and $\overline{dst}$ are the

Table 1 Tabular specification of a stack

Name	Guard	Input	Output	Assignment
pushItem	true	$e \cong push(DATA)$	$a' = \varepsilon$	$st' = list(DATA,st)$
getItem	not(isE(st))	$e \cong get$	$a' = ft(st)$	$st' = st$
popItem	not(isE(st))	$e \cong pop$	$a' = \varepsilon$	$st' = rt(st)$
idle	true	$e \cong \varepsilon$	$a' = \varepsilon$	$st' = st$

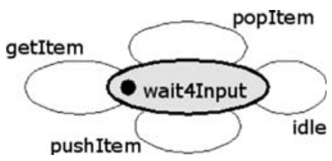

Fig. 1 STD of the stack

source and destination control states of the transition, respectively. By convention, we will use overlines for the names of control states. State machines are graphically represented by *state transition diagrams* (STDs) – circles (control states) and arrows (transitions). Two examples of STDs are given in Figs. 1 and 3. The black dot denotes the initial state, and only the labels of transitions are provided.

Each state machine is a rule system, but not every rule system is a state machine. However, there are many ways of transforming a rule system into a state machine. The simplest one is as follows: we add *state* of type $\{\bar{s}\}$ to L, add the conjunct $state = \bar{s}$ to the guard of each transition, and add the conjunct $state' = \bar{s}$ to the assignment of each transition (assuming $state \notin L$; otherwise we rename the old variable *state* before introducing the new one). Different ways of computing state machines from rule systems are the topic of this paper.

A *table* is the textual representation of a rule system in some tabular form. Parnas has devoted considerable work to the classification of tables [32]. For us, any tabular representation will do. An example of a table is given in Table 1.

2.3.1 Example

Consider the specification of a stack of integers. We assume a component with one input port $I = \{e\}$ with $type(e) = \{push(Int), get, pop, \varepsilon\}$, and one output port, $O = \{a\}$ with $type(a) = Int \cup \{\varepsilon\}$. There is one local variable, $L = \{st\}$. Using functional notation, its type is recursively defined by `data d_st = empty | list (Int, d_st)`. Three functions are defined: `isE(X) = (X == empty)`, `ft(list(X,Y)) = X`, and finally `rt(list(X,Y)) = Y`. Adding a further local variable *state* of $type(state) = \{\overline{wait4Input}\}$ to the set L of local variables generates a state machine from the rule system if trivial statements $state = \overline{wait4Input}$ and $state' = \overline{wait4Input}$ are simultaneously added to guard and assignment of each row of Table 1. Note the use of one transition-local variable, namely `DATA` in transition *pushItem*. Figure 1 shows the STD that corresponds to the state machine of the stack example.

3 Incremental development

Increments denote different development stages of a system, or model, respectively. To be as flexible as possible, we do not impose any constraints – such as the requirement that the set of all traces becomes smaller with each step as in stepwise refinement – on these steps.

3.1 Development process

Our experience with building large models boils down to the following process. Existing (informal) requirements specifications are read: a first understanding of the system's behavior is gained. One is capable of writing down statement such as "if a certain input occurs under certain conditions, then the system's state changes as follows, by outputting certain values". These rules are preliminary in that they are likely to be corrected later on. Reading the requirements documents also tends to lead to a first natural partitioning of the state space; for instance, one might find it natural to have a partitioning into *on* and *off* states in the model of an embedded system.

We found it expedient not to exclusively use the graphical STDs in these early stages of development. Instead, tables turned out to be tremendously useful. The reason is that we felt editing tables was easier than editing STDs. For instance, when adding transitions, arrows and labels have to be placed so they do not overlap too much with other states or transitions (contrast this with adding a row to a table). As a second example, when identical guards of several transitions have to be changed, the respective windows have first to be opened in the CASE tool (contrast this with directly pointing to the column or cells that contain the guards). As a third example, when several guards had to be checked for mutual exclusion, the respective windows had to be opened in the CASE tool (contrast this with having the respective guards automatically aligned). We are of course aware that this assessment is subjective, and that it also depends on the GUI of the CASE tool.

Nonetheless, there is no doubt that STDs are highly useful. Debugging is sometimes easier with executable STDs than with tables. For demonstration purposes with customers and domain experts, we found STDs to yield a good basis for discussion. In addition, the graphical layout helps one to identify symmetries, or missing symmetries which lead to corrections of the model (Sect. 6).

3.2 Modifications and refactorings

Development steps can alter interfaces, or they alter the behavior. We do not consider architectural modifications such as the addition of components here [7,34,39]. *Interface modifications* add or delete input or output ports from a system. If, before deletion, the name of a port does not occur in a system's transitions, its removal does not change the system's behavior, and neither does the introduction of a new port. *Behavior modifications* consist of removals and additions of traces of a model. Syntactically, this is achieved by inserting, modifying, or deleting transitions in T, possibly by taking into account modifications of L.

An increment $\tilde{R}$ of a rule system R with $[\![R]\!] \overset{I\cup O}{=} [\![\tilde{R}]\!]$ is called a *refactoring* of R. This assumes that R and $\tilde{R}$ define the same external interface $I = \tilde{I}$ and $O = \tilde{O}$: refactorings do not modify the interface of a component. An increment that is no refactoring is called a *modification*. In our incremental development process that relies on both tables (rule systems) and STDs (state machines), there are hence four different kinds of development steps that complement architectural and interface modifications:

1. refactorings of state machines,
   $\rho_S \in \{\rho \mid [\![R]\!] \overset{I\cup O}{=} [\![\rho(R)]\!]$ *and R is a state machine*$\}$,
2. refactorings of rule systems,
   $\rho_R \in \{\rho \mid [\![R]\!] \overset{I\cup O}{=} [\![\rho(R)]\!]$ *and R is a rule system*$\}$,
3. modifications of rule systems, and
4. modifications of state machines. Modifications modify, add, or delete transitions, possibly with alterations of L.

Let τ and τ^{-1} denote behavior-preserving transformations from rule systems into state machines, and vice versa. Figure 2 illustrates the relationship between the development steps (modifications denoted by δ). As development progresses from top to bottom, modifications take place. Within each row, different refactorings of both tables and STDs are considered, and the former can be transformed into the latter, and vice versa. The case study presented in Sect. 6 illustrates how this abstract process model is instantiated in practice.

In the next section, we will describe how to compute refactorings of rule systems, ρ_R. Since state machines are rule systems, this also caters for refactorings of state machines, and STDs, respectively. However, for reasons that we will be able to explain only after refactorings have been made precise, it is not always desirable to let $\rho_S = \rho_R$ (Sect. 5.3).

Refactorings of rule systems that are not state machines appear to be of moderate value: they remain textual, and we have discussed the benefits of graphical representations in Sect. 1. Methodologically, one

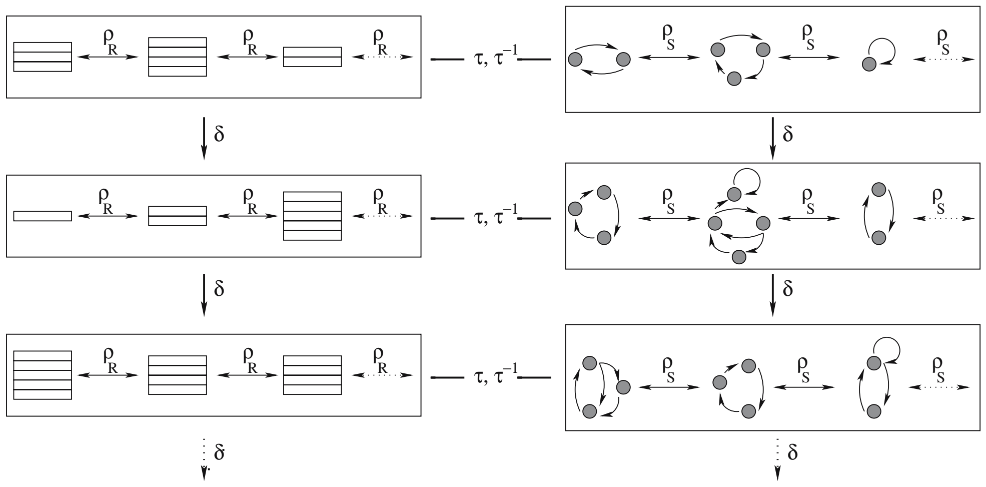

Fig. 2 Incremental development

would prefer to get a state machine (in fact, an STD) from a refactored rule system (in fact, a table) in one step. Consequently, we will focus on combinations of (1) refactorings of rule systems (tables) and (2) transformations from rule systems (tables) into state machines (STDs). As we will see in the next section, it is sufficient to consider refactorings of state machines defined by $\rho_S = \tau \circ \rho_R \circ \tau^{-1}$. The only reason for having included refactorings of rule systems into the left part of Fig. 2 is precisely that we compute refactorings of state machines by relying on these ρ_R.

4 Refactorings

This section constitutes the core of the article. We first discuss methodological considerations of model refactorings, and then present our approach to actually computing refactorings. We do not discuss the practically utterly relevant topic of co-evolution of models and code here [31, Sect. 3.4].

4.1 Methodological aspects of model refactorings

4.1.1 Refactorings of models and code

Refactorings for code tend to be motivated by a need for cleaning it up. Exactly in line with the process schematically depicted in Fig. 2, incremental code development proceeds by interleaving phases of adding functionality and cleaning up the code [17, p. 54]. In other words, when applied continuously, refactorings predominantly serve as a preparatory step for modifications. This is the case for both code and models, as exemplified in Sect. 6. However, code refactorings are mainly motivated by the notion of "code smells", i.e., "ugly" or redundant code portions. While we clearly acknowledge the need for refactorings when introducing hierarchical states, for instance, we have not encountered a comparable phenomenon of "model smells" with STDs. Instead, when we felt a need for refactorings, this was motivated by inadequate or "unnatural" control state structures, a result of an increasing understanding of the requirements. One reason for this difference possibly is the strongly restricted syntax, or conceptual simplicity, of STDs (without the action language) when compared to that of object-oriented code – there only is a restricted number of ways that STDs can look "ugly".

4.1.2 Patterns

Likely a consequence of the restricted syntax, too, there do not seem to be many refactoring patterns [17] for the circle-and-arrow part of STDs. One obvious reason is that our notion of refactoring always requires application-specific, or model-specific, knowledge as expressed in the predicates that define the transformation (Sect. 4.2). This, by definition, is somewhat contrary to the very idea of patterns. Still, even though the predicates have to be defined manually, one recurring situation is the desire to split one control state into several substates, which is comparable to making the state hierarchical (but for the sake of simplicity, our formalisms do not contain constructs for expressing hierarchies). In the case study in Sect. 6, all but one refactoring steps fall into this category. A dual situation occurs if one wants to merge several states into just one, and hence to get rid of hierarchy. The observation that these are the most frequent refactorings is interesting from an implementation perspective: in the splitting scenario, refactorings are "local" in that only those transitions of the original model need to be considered that lead to or emanate from the state to be split. Similarly, in the merging situation, only those transitions need to be considered that lead to or emanate from one of those states that are to be merged.

4.1.3 Reviews

Finally, it is precisely the restricted graphical syntax of STDs that makes them amenable to a certain kind of reviews. Symmetries, or more often a lack of symmetry of transitions between certain control states, can provide hints at incorrect or missing transitions (for instance, in non-hierarchical STDs, a missing transition from all states to the initial state that switches off a device). While "clean" code clearly facilitates reviews, the simple circle-and-arrow nature of STDs makes symmetry considerations particularly appealing. An example for this use of refactorings is provided in Sect. 6.8.

4.2 Intuition: refactoring via predicates

We are now ready to show how refactorings can be computed. In our stack example, one might want to transform the specification into an equivalent one with two control states: one specifies that the stack is empty, and the other one specifies that it is not. The problem then consists of computing the transitions between these two control states.

In this article, the idea of refactoring state machines or rule systems is to define a set of predicates that cover or even partition the data space (the case of predicates that do not form a covering is discussed in Sect. 4.5, and covering predicates that do not form a partitioning are handled in Sect. 4.6). In general, whether a set of predicates forms a partitioning or covering is undecidable. In our concrete case studies, however, we could easily see whether or not this was the case. Each of the predicates corresponds to one control state of the refactored model: control states are projections of the data space (defined as the set of all possible valuations of all variables). Once the covering predicates have been defined, one must compute the transitions between the corresponding states.

To get an intuition of this computation, consider a set of predicates, P, that cover the data space, and that do not constrain input nor output values. The elements of P will form the control states of the refactored model. Let $p, q \in P$. Transitions (arrows in the graphical representation) from p to q for each pair p, q are computed as follows. For each guard g of a row in the table, we compute the intersection between p and g, i.e., $p \wedge g$. We also need to make sure that q is compatible with the assignment $a \equiv \bigwedge_{l \in L} l' = f_l$ of the transition, i.e., that q holds if the assignment has been computed. Overall, the predicate $g \wedge p \wedge q'[f_l/l']_{l \in L}$ has to be satisfiable. With $|P|$ new control states and t transitions, the transformation requires the computation of $t \cdot |P|^2$ new transitions.

4.3 Example

In the stack example, suppose we want to derive a state machine with two control states characterized by the predicates $p \equiv isE(st)$ and $q \equiv not(isE(st))$. Clearly, p and q partition the data space. Table 2 shows the result of the refactoring where empty output ($a' = \varepsilon$) and trivial assignments ($st' = st$) are, for brevity's sake, omitted. Unsatisfiable transitions are canceled out.

For each transition of the original specification, four new transitions are computed: from p to p, from p to q,

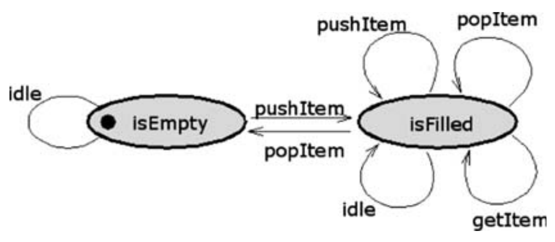

Fig. 3 STD of the refactored stack

from q to p, and from q to q. For instance, the first row in the table corresponds to a transition from p to p that is defined by the old transition *pushItem*. $isE(st)$ checks if the source control state, p, is compatible with the old guard, *true*. $isE(list(DATA, st))$ checks if the destination control state, p, is compatible with the old assignment, $st' = list(DATA, st)$. The conjunction of the two terms is unsatisfiable; the transition is canceled out.

As a second example, the tenth row of Table 2, marked by ††, is the transition from p to q w.r.t. the old transition *popItem*. $not(isE(st)) \wedge isE(st)$ checks the compatibility of the old guard, g, with the source control state, p. $not(isE(rt(st)))$ checks if the destination control state, q, is compatible with the old assignment. $p \wedge g$ are not satisfiable; this transition is also canceled out.

Figure 3 shows the STD of the stack as defined by Table 2 that we assume to be extended by the respective assignments to *state* and *state'*. Transitions are abbreviated. *isFilled* is the control state defined by $not(isE(st))$.

4.4 Formalization

We will now make the refactoring step precise. Let P denote a finite set of predicates over L, i.e., $\forall p \in P \bullet p \in \mathcal{T}(\hat{\Sigma}, L)$ for some extension $\hat{\Sigma}$ of Σ. P is required to

Table 2 Refactored behavior

Name	Guard	in: e≅	out: a′=	assgmt. st′=
~~pushItem~~	~~isE(st) ∧ isE(list(DATA,st))~~	~~push(DATA)~~		~~list(DATA,st)~~
pushItem	isE(st) ∧ not(isE(list(DATA,st)))	push(DATA)		list(DATA,st)
~~pushItem~~	~~not(isE(st)) ∧ isE(list(DATA,st))~~	~~push(DATA)~~		~~list(DATA,st)~~
pushItem	not(isE(st)) ∧ not(isE(list(DATA,st)))	push(DATA)		list(DATA,st)
~~getItem~~	~~not(isE(st)) ∧ isE(st) ∧ isE(st)~~	~~get~~	~~ft(st)~~	
~~getItem~~	~~not(isE(st)) ∧ isE(st) ∧ not(isE(st))~~	~~get~~	~~ft(st)~~	
~~getItem~~	~~not(isE(st)) ∧ not(isE(st) ∧ isE(st))~~	~~get~~	~~ft(st)~~	
getItem	not(isE(st)) ∧ not(isE(st)) ∧ not(isE(st))	get	ft(st)	
~~popItem~~	~~not(isE(st)) ∧ isE(st) ∧ isE(rt(st))~~	~~pop~~		~~rt(st)~~
††~~popItem~~	~~not(isE(st)) ∧ isE(st) ∧ not(isE(rt(st)))~~	~~pop~~		~~rt(st)~~
popItem	not(isE(st)) ∧ not(isE(st)) ∧ isE(rt(st))	pop		rt(st)
popItem	not(isE(st)) ∧ not(isE(st)) ∧ not(isE(rt(st)))	pop		rt(st)
idle	isE(st) ∧ isE(st)	ε		
~~idle~~	~~isE(st) ∧ not(isE(st))~~	~~ε~~		
~~idle~~	~~not(isE(st)) ∧ isE(st)~~	~~ε~~		
idle	not(isE(st)) ∧ not(isE(st))	ε		

$$\tilde{T} = \left\{ in \ \wedge \ g \wedge p \wedge q'[f_l/l']_{l \in L} \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o \,\middle|\, (in \ \wedge \ g \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o) \in T \ \wedge p, q \in P \right\} \quad (1)$$

$$\tilde{T} = \left\{ in \ \wedge \ g \wedge p \wedge q'[f_l/l']_{l \in L} \ \wedge \ state = \overline{p} \ \wedge \ state' = \overline{q} \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o \,\middle|\, (in \ \wedge \ g \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o) \in T \ \wedge \ p, q \in P \right\} \quad (2)$$

Fig. 4 Refactored transition relations

cover the data space L of a rule system $R = (V, S, T)$ with $V = I \cup O \cup L$ defined as above. P *covers* A_L iff for all states β, we have $\beta \models \bigvee_{p \in P} p$. For convenience, we also require that all predicates in P be satisfiable. Refactoring a rule system $R = (V, S, T)$ w.r.t. a covering P of the data space yields a rule system $\rho_R(R) = \tilde{R} = (V, S, \tilde{T})$ with $\tilde{T}$ being defined by equation 1 (Fig. 4, top). The proof that this transformation is indeed a refactoring, i.e., $[\![R]\!] \overset{I \cup O}{=} [\![\tilde{R}]\!]$, is given in Appendix A.

If one wants to perform the refactoring and generate a state machine in one step (Sect. 3), then the following construction can be used. With a new variable *state* of $type(state) = \bigcup_{p \in P} \{\overline{p}\}$ we define $\tau \circ \rho_R((I \cup O \cup L, S, T)) = (I \cup O \cup L \cup \{state\}, \tilde{S}, \tilde{T})$ with $\tilde{S} = S \wedge state = \overline{s}$ for some $s \in P$ with $S \Rightarrow s$, and $\tilde{T}$ being defined by Eq. 2 (Fig. 4, bottom).

4.5 Refinement

The reason for requiring the set of predicates to form a covering is as follows. Consider the stack example again, and assume a function *size* to be given. It computes the number of elements currently on the stack, $size(st)$. For some constant c, the predicates $\dot{P} = \{size(st) < c, size(st) = c\}$ do not form a covering of the stack's state space: the stack is bounded by a maximum number of c elements now. Computing a transformed stack on the grounds of Eq. (1) w.r.t. $\dot{P}$ then excludes transitions from and to states characterized by $size(st) > c$. In this sense, a refinement is computed: the set of traces of the original model is reduced, and the transformation is hence no refactoring.

In the general case, let Q denote a predicate (or set of predicates that are combined into one large disjunction) that describes the entire state space. Let U be any predicate (possibly a disjunction of predicates) that does not cover the state space. The "missing parts" of Q can then be described by a predicate M with $\neg(M \wedge U)$, such that we have $(U \vee M) \Leftrightarrow Q$. Computing Eq. (1) w.r.t. U rather than the covering predicate Q means that the transformed system does not contain any transitions from or to the states represented by M. The transformation is a refinement rather than a refactoring.

In case M is known (which implies knowing that $U \vee M$ forms a covering), this can hence methodologically be exploited for computing refinements. In case M is not known (which implies not knowing whether or not U forms a covering), a computation w.r.t. Eq. (1) means that inadvertently a refinement rather than a refactoring may be computed.

4.6 Internal Nondeterminism

The proof that the construction of a new set of transitions defined by Eq. (1) leads to a refactored rule system only requires P to cover but not to partition the state space. In addition to covering the state space, the predicates in a *partitioning* predicate set P must be pairwise disjoint, i.e., $\forall p, q \in P \bullet p \not\Leftrightarrow q \Rightarrow \neg(p \wedge q)$. Choosing P to be a partitioning ensures that no internal nondeterminism is introduced.

Consider refactoring the stack w.r.t. three predicates $p_1 \equiv size(st) = 0$, $p_2 \equiv size(st) \in \{1, \ldots, c\}$, and $p_3 \equiv size(st) \geq c - 2$ for some constant $c > 2$. The state space is clearly covered, but p_2 and p_3 overlap in that $p_2 \wedge p_3$ is satisfiable. Consider the respective state machine with control states $\overline{p_1}$, $\overline{p_2}$, and $\overline{p_3}$, and a partial execution with the system being in $\overline{p_2}$ with $size(st) = c - 1$. A *push* command can make the system either remain in state $\overline{p_2}$, or transfer control to $\overline{p_3}$. The proof ensures that the externally visible behaviors remains identical (it does not take into account the explicit state variable). This situation is exemplary for a choice of predicates that cover yet do not partition the state space. As shown in Appendix B, our transformation does not introduce this kind of nondeterminism whenever actual partitionings are taken as a basis for the refactoring step.

5 Implementation

As far as we know, there is no model-based CASE tool that integrates tables and STDs. We have used Excel and AutoFocus with ad-hoc translations between the two. While not yet integrated into the tool, the computation of refactorings is automated and includes (a) the – trivial – computation of refactored transitions (set $\tilde{T}$), and (b) their – non-trivial – simplification, possibly to *false*, which we will describe below. Step (b) is particularly important because the computed transitions should be readable by humans, and, as the example of Sect. 4.3 shows, there is a great potential for the removal of redundant parts.

The subject of this section is simplification. We describe a simplification algorithm that includes a simple satisfiability checker (Sect. 5.1) and implements the rules of Boolean algebra (Sect. 5.2). The former is used to remove unsatisfiable disjuncts for formulas in disjunctive normal form. When computing refactored STDs from tables, the construction of Sect. 4.4 introduces a new *state* variable for each such refactoring step. As it turns out, many of these variables and references to them can be deleted, which is explained in Sect. 5.3.

5.1 Satisfiability checking

Because of potentially infinite data structures, the satisfiability problem is generally undecidable, but one could argue that (a) the cut-off of infinite data structure that can often be justified by domain knowledge, and (b) the simplicity of the involved functions – e.g., there is usually no mutual recursion, and most recursions turn out to be primitive, i.e., terminating – make manual decisions possible. Because our action language for guards and assignments is a functional language, we have implemented the simplifier in the functional logic language Curry [10]. Curry's operational semantics relies on narrowing [19] which explains why it lends itself to satisfiability checking.

The complexity of a satisfiability check obviously depends on how contrived the involved functions and state characterizations are. With a restriction of all lists to a maximum length of 5, the examples in Sect. 6 are computed in negligible time (the problem is exponential in the length of the involved lists), and all stack examples are computed in negligible time without depth restrictions. We have not implemented a plugin that also takes into account automatic layouting of computed STDs.

We will assume that a system has n input ports $ci_1, \ldots, ci_n$, m output ports $co_1, \ldots, co_m$, and w local variables $l_1, \ldots, l_w$. Every transition t given by

$$\bigwedge_{j=1}^{n} ci_j \cong i_j \wedge g \wedge \bigwedge_{j=1}^{w} l'_j = f_{l_j} \wedge \bigwedge_{j=1}^{m} co'_j = f_{o_j} \tag{3}$$

directly translates into a Curry function *step* defined by

$$\begin{aligned} &step\ t\ (l_1, \ldots, l_w)\ (i_1, \ldots, i_n) \mid g \\ &\quad = \big((f_{o_1}, \ldots, f_{o_m}), (f_{l_1}, \ldots, f_{l_w})\big), \end{aligned} \tag{4}$$

with the intuitive semantics that whenever actual parameters match the formal parameters expressed by the patterns t, l_i, and i_j, and in addition, guard g evaluates to true, then the function returns a pair that consists of output values and updated local variables.

The functional-logic, lazy-evaluation and higher-order nature of Curry enables one to check if a transition from p to q (with $p, q \in P$ being predicates that characterize the new control states) is compatible with a previously existing transition t by simple function application. Compatibility here means satisfiability of an item of the set $\tilde{T}$ as defined by Eq. (1), and it is checked by the simple program

```
sat p q t = p L && (q (snd (step t L I)))
                                                 where L,I free
```
(5)

where p and q are the programs that characterize source and destination states, t ranges over all possible transitions, L denotes the vector of all local variables, and I is the input. L and I are computed automatically by the Curry runtime system. *snd* computes the second element of a pair.

Since L is a vector of variables, or their valuations, repectively, one can relate `L` and predicates `p` by means of the satisfaction relation, $\models$. Program 5 returns a solution only if $L \models p$. This is achieved by applying the function p to L. Furthermore, by calling the *step* function, it indirectly ensures $L \models g$ because the guard is evaluated there, and also $L' \models q'$ because this is what is encoded in the second element of the return value of the *step* function.

5.1.1 Example

Consider the stack example again. We will implement the stack by means of a list (defined by constructors `[]` and `:`, and with accessor functions `ft` for the first element and `rt` for the everything but the first element). Empty output is denoted by `oeps`. Without going into the syntactic details of Curry (`\V -> F` denotes lambda abstraction $\lambda v.f$, `findall` enumerates all solutions of a given equation, and `<-` denotes list comprehension), the following program is enough to compute all those transitions between the empty and the non-empty states that are satisfiable. This is done by simply executing function `checkall` defined below.

```
step getT  st  getI       | False=:=isE st =
                            (o (ft st), st     )
step pushT st (pushI N)    = (oeps,      N:st   )
step popT  st  popI       | False=:=isE st =
                            (oeps,      rt st)
isE []    = True
isE (_:_) = False
p = isE
q = not isE
allstates = [p,q]
sat p q T = p L && q (snd (step T L I))
                         where L,I free
feasible start stop =
     findall ((\T -> (sat start stop T)=:=True)))
checkall  = [(start,stop,feasible start stop) |
               start<-allstates, stop<-allstates]
```

5.2 Simplification

For more complex problems, however, this is too simple to work. We have also implemented a simplifier that takes care of simplifying propositional formulae w.r.t. the standard laws of Boolean algebra (among others, for instance, $a \wedge \neg a$ is simplified to *false*, etc.). Because these laws are standard, and the implementation of such a simplifier is standard as well, we do not show the implementation. One remark, however, is in order. To avoid unnecessary computations, the above definition of the `sat` function does contain calls to the simplifier. However, one cannot simply define

```
sat p q T = simplify (p L && q (snd (step T L I)))
                                    where L,I free,
```

which contains such a reference. This is because `simplify` would then have to operate on (partially) applied functions that are created during the computation. However, access to such functions is difficult to achieve, because the formal parameters of function definitions have to be constructor terms and must not contain function symbols (constructors are uninterpreted symbols, such as `oeps` in the above example, and function symbols are interpreted by the right hand side of the defining functions, such as `isE`). For instance, to implement the equality $\neg\neg a = a$, one cannot simply define `simp (not (not a)) a = a` because function symbols like *not* as patterns are not allowed. Our solution to this problem consists of first generating pure constructor terms, i.e., terms without function symbols, by replacing all function symbols with dedicated constructors. Simplification is then performed on these constructor terms, and after simplification, the constructor terms are re-translated into the respective function symbols.

5.3 Removing the *state* variable

Assume an iterative process where a state machine, or an STD, is generated, modified, re-transformed into a table which is subsequently modified, etc. Adding a new *state* variable for each transformation from a rule system to a state machine (Eq. (2) in Sect. 4.4) is likely to clutter the model or, more precisely, the guards and assignments of transitions. This is the only reason for not letting $\tau^{-1} = id$ (Sect. 3). It is not a conceptual but rather a practical problem: we would like the rule systems to be readable by humans, and thus contain as little redundancy as possible.

It turns out that many modifications are of a nature that makes it possible to remove previously introduced *state* variables. We will now identify those development steps that, when applied in-between two refactorings, allow one to delete references to the *state* variable introduced in the last refactoring step. As explained in Sect. 3, we focus on behavior modifications and ignore interface modifications.

Recall that the computation of a refactoring and a state machine in one step, defined in Eq. (2) in Sect. 4.4, is done for each transition with assignment $\bigwedge_{l \in L} l' = f_l$ and each pair of predicates, p and q. By construction, we have $state = \overline{p}$ if p holds. Conversely, we have $state' = \overline{q}$ if $q'[f_l/l']_{l \in L}$ evaluates to true. In other words, the information on the explicit *state* variable is redundant; it can be synthesized from p or $q'[f_l/l']_{l \in L}$, respectively. At this stage it is only used to decide whether or not to draw a transition arrow between two control states of the respective STD.

In the following, we assume that a CASE tool maintains some representation of a given state machine and is able to display both representations, table and STD. Each row of the table corresponds to one arrow in the STD. Modifications can take place both at the level of the table and the level of the STD. We will now take a look at possible development steps between two refactorings (i.e., introductions of *state* variables), and how these relate to the necessity of keeping references to earlier introduced *state* variables in the generated transitions. These steps are removal, insertion, and modification of a transition, addition and removal of control states, and addition or removal of local variables. The following list is to be read as follows. If the modifications between two refactorings are all described by items that mention that the *state* variable introduced in the first refactoring step carries redundant information only, then all references to it can be removed directly before the second refactoring step.

1. *Removing* transitions from the STD or corresponding rows from the table after a refactoring from a rule system into a state machine is not problematic: the *state* variable from the first refactoring step carries nothing but redundant information.

2. *Adding* a new transition in an STD from an existing control state $\overline{p}$ to an existing control state $\overline{q}$ boils down to two different situations. Assume that the new transition's assignment is $\bigwedge_{l \in L} l' = f_l$, and that the set L of local variables was not changed.
   - The *state* variable introduced in the first refactoring step carries redundant information only if $q'[f_l/l']_{l \in L}$ is satisfiable: no inconsistency with the logical characterization of the destination state $\overline{q}$ is introduced. The CASE tool simply has to add $state = \overline{p} \wedge p \wedge q'[f_l/l']_{l \in L}$ to the guard and $state' = \overline{q}$ to the assignment of the table's row that corresponds to the new transition. In the sequel, it can be treated like any other transition; the *state* variable carries redundant information only.
   - On the other hand, if $q'[f_l/l']_{l \in L}$ is not satisfiable, then the new transition violates the logical characterization of $\overline{q}$. In this case, the *state* variable introduced in the first refactoring step carries actual information and cannot be deleted before the second refactoring step. An example of this situation is given in the case study in Sect. 6.5.2.

   Adding a new row to the table is independent of any information associated with the *state* variable, unless, of course, it explicitly references this variable.
3. *Modifying* a transition can be seen as the process of removing a transition and then adding a new one. This case is hence covered by items 1 and 2 above.
4. *Adding a new control state* $\overline{r}$ (and hence extending the type of variable *state* by $\overline{r}$) in itself is obviously not problematic. However, if there is no logical characterization of $\overline{r}$, and new transitions from or to $\overline{r}$ are added, then the *state* variable in the tabular representation of this new transition does not carry redundant information alone. It cannot be deleted before the second refactoring step (even though it might be the case that it is relevant only for transitions from or to $\overline{r}$).
5. *Removal* of control states usually entails the deletion of all incoming or outgoing transitions. In case there are such transitions, this is covered by item 1 above. If there are no such transitions, then no transition in the model contains references to the state that is to be deleted: the *state* variable contains redundant information only.
6. Finally, modifications of $L - \{state\}$ can be either removals or additions of variables. The *addition* of variables in itself obviously is not problematic. The *removal* of variables requires a modification of the logical characterization of the control states as well as of guards, assignments, and output statements: these must not contain any references to the removed variable. For each transition in the model, it is also necessary to perform an analysis as described in step 2 above.

In other words, if the CASE tool implements the checks as described above, and if it can be decided that for a particular set of subsequent development steps, a previously introduced *state* variable carries nothing but redundant information, then it can be removed in a subsequent refactoring step, which leads to simplified rules.

6 Example: MOST NetworkMaster

This section illustrates the methodological benefits of our approach when applied to the behavior model of a network controller for automotive infotainment systems, the MOST NetworkMaster (NM) [30]. The model was the basis for applying and assessing model-based testing technology in an NM implementation [38].

The functionality of the infotainment network is divided into function blocks which reside on the network's devices. Examples include starting the CD player, or displaying a video stream on one of the displays. The NM is a special function block responsible for network management. In this article, we consider only the model of the NM's main service: setting up and maintaining the *central registry*. The central registry contains all function blocks and their associated network addresses currently available in the network.

In the following, we apply our refactoring techniques to the incremental development of a slightly simplified NM model.

6.1 Overview

We start by defining the syntactic interface of the system in Sect. 6.2. Section 6.3 describes the initial increment, the startup phase of the controller. In Sect. 6.4, we apply an intuitive refactoring into the states *on* and *off*. The next increment, defined by the handling of problematic devices during the startup phase, is described in Sect. 6.5. This step consists of two substeps, one preparatory refactoring step, and the actual increment. Section 6.6 suggests a further refactoring that appears expedient, and that is a result of insights from earlier steps. Section 6.7 describes the final model, and Sect. 6.8 demonstrates the usefulness of refactorings when used for reviews. It also contains an example of a refactoring where it is not the case that two states are merged into one new state, or where one state is split into two new states (cf. Sect. 4.1).

Table 3 First increment of the NM model

Name	Guard	in: (net,i)≅	out: o′=	Assignment
NetOn	isE(al)	NetOn(N), ε	ε	al′=init(N) ∧ reg′= empty ∧ wa′=empty
NetOff	not(isE(al))	NetOff, ε	ε	al′=empty ∧ reg′= empty ∧ wa′=empty
sndFBGet	not(isE(al)) ∧ isE(wa) ∧ not(allReq(al))	ε, ε	FBGet(nxtAddr(al))	al′=setReq (nxtAddr(al), al) ∧ wa′=nxtAddr(al)
recFBStatus	not(isE(al)) ∧ wa=[ADDR] ∧ not(allReq(al))	ε, FBStatus(ADDR, FBL)	ε	reg′=store(ADDR, FBL, reg) ∧ wa′=empty
recFBStatus SndOK	not(isE(al)) ∧ wa=[ADDR] ∧ allReq(al)	ε, FBStatus(ADDR, FBL)	CfgStatusOk	reg′=store(ADDR, FBL, reg) ∧ wa′=empty

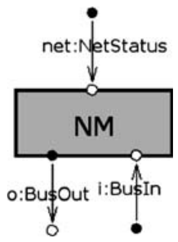

Fig. 5 Interface of the NM model

6.2 Black-box view

Initially, the interface of the model of the NM consists of the input ports $I = \{net, i\}$ and output port $O = \{o\}$. *net* carries signals for switching on and off the network, and *i* carries signals for incoming network messages. *o* carries outgoing network messages (Fig. 5).

6.3 Step 1: Startup

We start by modeling the startup behavior of the NM. Whenever the network is switched on, the NM requests the function blocks of each device in the network, and stores them in the central registry. Afterwards, the NM sets the network to normal operation by broadcasting a message indicating that the network's configuration status is okay. This means that the devices in the network are allowed to freely communicate and use each other's function blocks.

For the first modeling step, we make use of three local variables *al*, *wa*, and *reg*.

- *al* stores the list of network addresses of the devices in the network, and attaches to each address a flag that indicates whether or not that address has already been contacted during startup. The NM asks each of these addresses to return their function blocks.
- *wa* stores the network address from which the NM expects an answer to its last request. It is implemented as a list that contains at most one element.
- *reg* holds the actual central registry.

All variables are initialized by their default value *empty*. Table 3 shows the first increment of the NM model with five rules.

1. Via rule *NetOn*, the network is switched on, and the address list *al* is initialized to the number *N* of devices in the network.
2. Via rule *NetOff*, the network is switched off, and all local variables are set to their default values.
3. Rule *sndFBGet* encodes the request of function blocks from the next network address. If there are further addresses to request (predicate *not(allReq(al))*) and the NM does currently not wait for an answer (predicate *isE(wa)*), then the next address is requested and this address is recorded (assignment of variable *al*).
4. Rule *recFBStatus* models the reception of an answer. This answer is stored in the central registry (assignment of variable *reg*).
5. Finally, rule *recFBStatusSndOk* models the reception of the last answer and broadcasting a message that the configuration procedure is done (message *CfgStatusOk*). The NM thus sets the network to normal operation; all devices are allowed to communicate freely.

6.4 Step 2: Refactoring

It appears intuitive to partition the state space by means of the predicates $on \equiv not(isE(al))$ and $off \equiv isE(al)$,

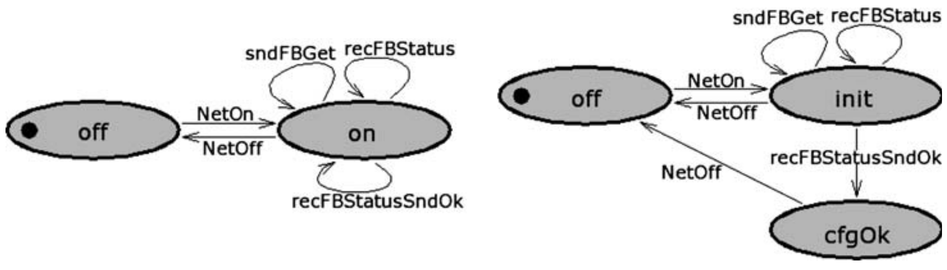

Fig. 6 Two variants of the first increment

Table 4 Refactoring of the first increment of the NM model (cf. Fig. 6, right)

Name	Guard	in: (net,i)≅	Assignment
NetOn	isE(al) ∧ not(isE(init(N))) ∧ not(allReq(init(N))) ∧ state=$\overline{\text{off}}$	NetOn(N), ε	al′=init(N) ∧ reg′=empty ∧ wa′=empty ∧ state′=$\overline{\text{init}}$
NetOff	not(isE(al)) ∧ (not(allReq(al)) ∨ not(isE(wa))) ∧ state=$\overline{\text{init}}$	NetOff, ε	al′=empty ∧ reg′=empty ∧ wa′=empty ∧ state′=$\overline{\text{off}}$
NetOff	not(isE(al)) ∧ allReq(al) ∧ isE(wa) ∧ state=$\overline{\text{cfgOk}}$	NetOff, ε	al′=empty ∧ reg′=empty ∧ wa′=empty ∧ state′=$\overline{\text{off}}$
sndFBGet	not(isE(al)) ∧ isE(wa) ∧ not(allReq(al)) ∧ not(isE(setReq(nxtAddr(al),al))) ∧ (not(allReq(setReq(nxtAddr(al),al))) ∨ not(isE(nxtAddr(al)))) ∧ state=$\overline{\text{init}}$	ε, ε	al′=setReq(nxtAddr(al),al) ∧ wa′=nxtAddr(al) ∧ state′=$\overline{\text{init}}$
~~sndFBGet~~	~~not(isE(al)) ∧ isE(wa) ∧ not(allReq(al)) ∧ not (isE(setReq(nxtAddr(al),al))) ∧ allReq(setReq(nxtAddr(al),al)) ∧ isE(nxtAddr(al)) ∧ state=init~~	~~ε, ε~~	~~al′=setReq(nxtAddr(al),al) ∧ wa′=nxtAddr(al) ∧ state′=cfgOk~~
recFBStatus	not(isE(al)) ∧ wa=[ADDR] ∧ not(allReq(al)) ∧ state=$\overline{\text{init}}$	ε, FBStatus(ADDR, FBL)	reg′=store(ADDR,FBL,reg) ∧ wa′=empty ∧ state′=$\overline{\text{init}}$
recFBStatusSndOK	not(isE(al)) ∧ wa=[ADDR] ∧ allReq(al) ∧ state=$\overline{\text{init}}$	ε, FBStatus(ADDR, FBL)	reg′=store(ADDR,FBL,reg) ∧ wa′=empty ∧ state′=$\overline{\text{cfgOk}}$

and perform a respective refactoring. The transformation of Table 3 then results in the state machine depicted in Fig. 6, left.

6.5 Step 3: missing devices

In the next development step, we model the situation where some of the devices do not respond to the NM's request at the first time. The NM requests these pending devices after it has set the network to normal operation.

6.5.1 Preparation: refactoring

Before we model this behavior, it is expedient to prepare the model for this development step by splitting state *on* $\equiv$ *not*(*isE*(*al*)) into two states, namely *init* $\equiv$ *not(isE(al))* $\wedge$ *(not(allReq(al))* $\vee$ *not(isE(wa)))* and *cfgOk* $\equiv$ *not(isE(al))* $\wedge$ *allReq(al)* $\wedge$ *isE(wa)*. Obviously, *on* $\Leftrightarrow$ *init* $\vee$ *cfgOk* holds. Application of the splitting transformation described in Sect. 4.1.2 yields the state machine depicted in Fig. 6, right. By doing so, we have refactored the model into a state machine where the status *network has reached normal operation* is explicit in the model. The resulting state machine is described in Table 4 (with output statements omitted, for the sake of brevity; cf. Table 3). Note that in this table, the explicit *state* variable is redundant and could be deleted without changing the semantics. The second *sndFBGet* transition, from state *init* to state *cfgOk* is never enabled, a consequence of the fact that the function definitions entail *not*(*allReg*(*al*)) $\Rightarrow$ *not*(*isE*(*nxtAddr*(*al*))). The unsatisfiability of all those transitions that are omitted from the table directly follows from the laws of Boolean algebra and intuitively true statements such as *isE(empty)*.

6.5.2 Increment

Now we are ready to model the new behavior described above. We start by adding a further flag to the list of addresses, *al*. This flag is set when the respective devices have answered. This is implemented by a function *setAns*, as shown in Table 5. This table does not repeat

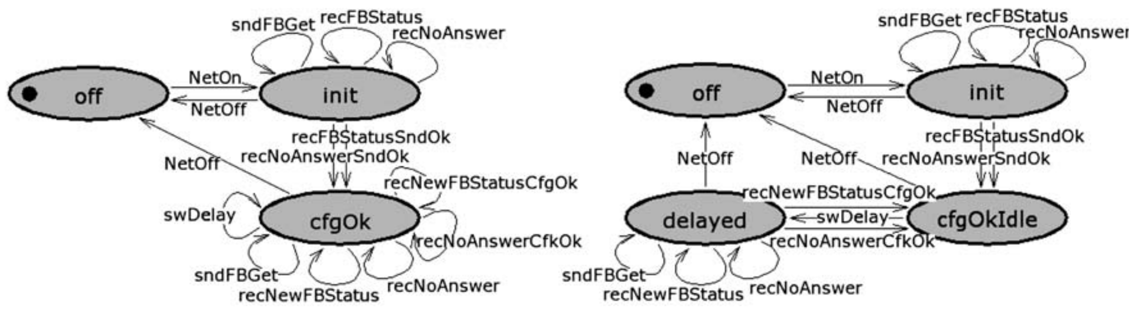

Fig. 7 Two variants of the second increment

Table 5 Incomplete second increment (cf. Figure 7, left)

Name	Guard	in: (net,i)≅	out: o′=	Assignment
recFBStatus	not(isE(al)) $\wedge$ wa=[ADDR] $\wedge$ not(allReq(al)) $\wedge$ state=$\overline{\text{init}}$	ε, FBStatus (ADDR, FBL)	ε	**al′=setAns(ADDR,al)** $\wedge$ reg′=store(ADDR,FBL,reg) $\wedge$ wa′=empty $\wedge$ state′=$\overline{\text{init}}$
recFBStatus SndOK	not(isE(al)) $\wedge$ wa=[ADDR] $\wedge$ allReq(al) $\wedge$ state=$\overline{\text{init}}$	ε, FBStatus (ADDR, FBL)	CfgStatusOk	**al′=setAns(ADDR,al)** $\wedge$ reg′=store(ADDR,FBL,reg) $\wedge$ wa′=empty $\wedge$ state′=$\overline{\text{cfgOk}}$
swDelay	**not(isE(al))** $\wedge$ **isE(wa)** $\wedge$ **allReq(al)** $\wedge$ **state=$\overline{\text{cfgOk}}$**	ε, ε	ε	**al′=setReq2NotReq(al)** $\wedge$ **state′=$\overline{\text{cfgOk}}$**
sndFBGet	**not(isE(al))** $\wedge$ **isE(wa)** $\wedge$ **not(allReq(al))** $\wedge$ **state=$\overline{\text{cfgOk}}$**	ε, ε	**FBGet(nxtAddr(al))**	**al′=setReq(nxtAddr(al),al)** $\wedge$ **wa′=nxtAddr(al)** $\wedge$ **state′=$\overline{\text{cfgOk}}$**
...	...	...	...	...

the rules of Table 4, and for the sake of brevity, not all transitions that are shown in the STD of Fig. 7, left, are listed in the table. Modifications are typeset in boldface.

When the system is in state *cfgOk* and all requests have been issued, then it is necessary to take care of those addresses that have not answered yet. This is done by remaining in this control state, while setting the “requested” flag of all those addresses that have not yet answered to “not requested”, thus ensuring that they will be asked once more (rule named *swDelay*).

However, by doing so, the logical characterization of *cfgOk* is violated. Among other constraints, *cfgOk* is defined by *allReq(al)* (Sect. 6.5.1). This is an example of the situation described in Sect. 5.3, second bullet of item 2. Once this rule – or transition in the respective STD – has been added, one cannot simply remove the state variable introduced by an earlier refactoring step. The reason is that the definition of rule *swDelay* now depends on explicit information about the control state, namely *cfgOk*, and this information is not equivalent to the logical characterization of *cfgOk*, as we have seen.

A second consequence of this development step is that the developer needs to reconsider rules that were defined earlier. For instance, it is now necessary to add a rule that, *while being in state cfgOk*, sends requests to addresses from which no answer has been obtained yet. This is done by rule *sndFBGet* shown in Table 5.

The remaining functionality is implemented in the transitions that emanate from state *cfgOk*. While not all transitions are shown in the table, the complete state machine is depicted in Fig. 7, left.

6.6 Step 4: refactoring

After completion of the incremental development step, we can once more apply the mechanism from Sect. 4.1.2 in order to split states and separate a state called *delayed* from state *cfgOk*. The idea is that there are two modes, one indicating that all nodes have answered (state *cfgOkIdle* ≡ *state=cfgOk* ∧ *allReq(al)* ∧ *isE(wa)*), and the other one indicating that some nodes have not yet been requested once more (state *delayed* ≡ *state=cfgOk* ∧ *(not(allReq(al))* ∨ *not(isE(wa)))*). Note that the earlier introduced variable *state* is referenced. Figure 7, right, shows the respective STD; the table is omitted for the sake of brevity.

6.7 Completion

We do not show any further modeling details here. During modeling we identified five main modes of the NM which result in a variable called *mode*: in mode *off* the NM is switched off; in mode *init* the NM performs a system configuration check during startup – all devices are asked for their function blocks; in mode *cfgOk* the NM has set up the network to normal operation, i.e., all devices are allowed to communicate freely; in mode *ncd* the NM performs a system configuration check after a network change, i.e., a device has left or jumped in the network; and in mode *delayed* the NM requests periodically devices which have not answered to any request yet. In an advanced modeling stage the NM's service is specified by a table with 17 rows where most guards contain four or five atoms.

6.8 Refactorings for reviews

We transformed this table into different state machines for a review of the model. We chose the partitioning P_1 which divides the state space according to the five modes of the NM (Sect. 6.7). This is done by using an explicit *mode* variable. Figure 8, left, depicts the respective state machine. In addition, we chose a second partitioning P_2 which distinguishes between the following states: (1) *requestingDevices* $\equiv wa = empty \wedge mode \in \{init, ncd, delayed\}$ where the NM requests devices, (2) *waitForFBStatus* $\equiv wa \neq empty \wedge mode \in \{init, ncd, delayed\}$ where the NM waits for an answer, (3) *off* and (4) *cfgOk* where the NM is in modes *off* or *cfgOk*. Figure 8, right, depicts the state machine w.r.t. partitioning P_2 where "req.Devices" corresponds to predicate *requestingDevices*, and "wait4FBStat" to predicate *waitForFBStatus*. While the other refactorings of this chapter all deal with splitting one state into several new ones or vice versa, the refactoring w.r.t. P_2 shows how three states (*init, ncd, delayed*) are refactored into two states (*requestingDevices, waitForFBStatus*). Neither *requestingDevices* nor *waitForFBStatus* are pure mergers of two states defined by P_1.

P_1 allows us to study symmetries w.r.t. mode switching. For example, upon each network reset, the NM returns to mode *init* (transitions with names ending in *NotOk*). We would have detected an error in the model if one of these transitions had been missing. By means of P_2, we can observe that the NM can enter state *requestingDevices* from state *cfgOk* only if a network change occurs (transitions beginning with *NCD*) or if there are devices which have not answered yet (transition *swDelay*). There would be an error if there were further transitions. This example demonstrates that specific symmetries can be found and analyzed by building different abstract views of rule systems. By reviewing this kind of abstractions, the model can be analyzed easily if some transitions must or must not exist for symmetry considerations. The abstract view reveals relations in the model which would likely have stayed hidden in the detailed view of tables.

7 Related work

Related work can be structured into other approaches to refactoring, the use of tables, incremental development processes, and logical characterizations of state spaces. A preliminary version of this article appeared as a conference contribution [36].

Refactorings: Sunyé et al. consider the refactoring of statecharts on the grounds of hierarchical states [42]. Roughly, sets of states are merged, and the new transitions are computed. This differs from our work in that they do not consider *arbitrary* new definitions of states (our sets P that cover the state space); merging states is just one refactoring (Sect. 4.1.2).

In the context of inductive verification, Cheng considers refactoring a parameterized process into a set of constant processes [9]. In our context, this would amount to refactoring one state machine into more than one state machine, which is not the focus of this paper.

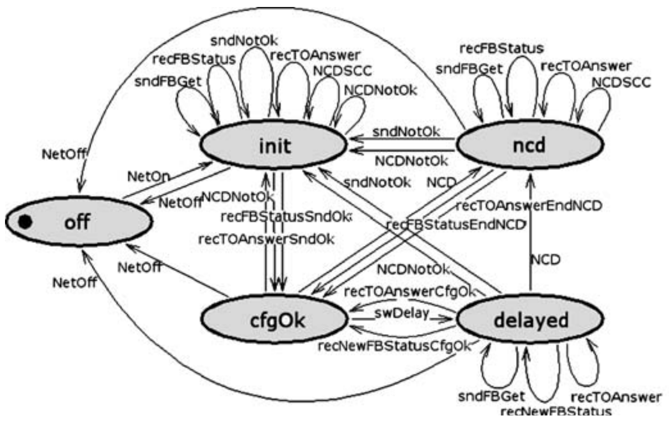

Fig. 8 Two variants of the last increment (via P_1: *left*; via P_2: *right*)

Van Gorp et al. [44] propose extensions to the UML meta model such that pre- and postconditions for behavior-preserving transformations can be expressed. This work is not concerned with refactorings of state machines. Similarly, Correa and Werner discuss refactorings of OCL expressions and class structures, without explicitly taking into account state machines [11].

Philipps and Rumpe [40] present a set of transformation rules for data flow networks and formally show that the transformed system is a refinement of the original one. Their work differs from ours in that we actually compute the refactoring of a behavior model.

At first sight, one may be tempted to see similarities between our work on refactorings and the ideas of predicate abstraction [18]. The commonality is that both approaches rely on predicates that define the states of a transformed system. The difference is that our transformation does not induce any loss of information, or abstraction, in the transformed system. This is the reason why we can – modulo simplification – compute the transitions of the transformed system by syntactic replacements only, which is in general not the case for the computation of abstract transitions in the context of predicate abstraction.

Tables: Shen et al. [43] are concerned with transformations of tabular specifications of a system. They concentrate on transformations between different kinds of tables [32] rather than transforming tables into graphical representations in the form of extended state machines. Their transformations are refactorings in their own right.

Incrementality: Prowell and Poore use incrementally discovered equivalence classes on I/O sequences to specify the I/O behavior of a system [35]. One could directly use such canonical sequences as states (we do not provide a method for deriving (characterizations) states but rather assume them to be given – note that the interpretation of states as equivalence classes on input histories is standard, as witnessed, for instance, by Nerode's classical right congruence used to minimize finite automata). Janicki and Sekerinski claim that this leads to complex state machines even for small systems [24]. In that latter paper, the trace assertion method is revisited, and by directly catering for certain signal interleavings, the authors propose to interpret certain so-called step-traces as states. Both approaches do not seem to see a need for refactorings at all, but they also advocate the use of different specifications.

Logical Characterization: The state invariants in timed and hybrid automata [1,27] are obviously related to our logical characterization of refactorings. However, we are concerned with discrete systems, and we use the invariants in a methodologically different manner, namely to the end of refactoring. Furthermore, state invariants in timed and hybrid systems need not cover the state space.

Lamport uses TLA predicates – invariants – to characterize control states [25] in predicate-action diagrams. Except for the concrete language, this is similar to what we do in this paper. However, Lamport is not concerned with refactorings.

Finally, the predicates we use to characterize control states relate to the "reaffirmed invariants" in the context of the Stanford Temporal Prover [28], namely local invariants $PC = i \Rightarrow I(i)$ that describe properties $I(i)$ at program location i and that are defined on data variables only. These special invariants are dubbed "mode invariants" in the SCR context [23].

8 Conclusions and future work

The starting point of our work is the observation that current model-based CASE tools provide insufficient support for the incremental development of STDs when it comes to fundamental changes of the control states. These might become necessary if a better understanding of the systems suggests a different, more adequate, perspective on the state space. Refactorings of STDs are hence motivated by a better understanding of the system rather than by a "model smell".

We have shown a way of computing refactorings of state machines on the grounds of predicates that describe parts of the state space. Our incremental development process is based on both tables and STDs. We have argued that there is room for both representations, and that it is beneficial to use them in parallel: because of their clear structure, tables are sometimes easier to grasp and manipulate – and STDs help with identifying symmetries and, possibly together with simulation traces in the form of sequence diagrams, also with conveying fundamental ideas behind the model. Refactoring tables that do not represent state machines appears to be of modest value. Benefits do become apparent when the simultaneous transformation into STDs is considered.

Because the computed refactorings are meant to be readable by humans, we have shown how refactoring steps can be performed with both representations while reducing to a minimum the number of conjuncts in guards that are introduced by the computation of a refactoring. We have described those development steps that allow one to remove previously introduced redundant information when a refactoring is done simultaneously with a transformation into an STD.

There are some limitations to the applicability of our approach. Firstly, because determining whether or not a set of predicates forms a partitioning is in general an undecidable problem, one might inadvertently compute a refinement rather than a refactoring (Sect. 4.5). In our experience, however, the covering property of a set of predicates could always be shown. Secondly, while the computation of a refactoring as defined in Sect. 4.4 is a purely syntactical transformation, the simplification of the resulting transitions – without, these computed transitions quickly become unintelligible – requires reasoning. With sufficiently powerful action languages, this in general also is an undecidable problem. Even in decidable cases, the performance of respective satisfiability checkers such as the one described in Sect. 5 naturally affects the practical utility of the approach. Thirdly, the definition of covering predicates is a challenging task that, in addition to intimate familiarity with the system under consideration, requires knowledge of formal logics that some modelers will have to gain before the computation of refactorings becomes as natural an activity as adding or removing behavior.

Our experience with behavior models of embedded systems that we built to the end of generating test cases suggests that the cost of building and maintaining the models is likely to turn out as a critical parameter. In many cases, the potential of considerable reuse will drive the decision for or against this or comparable technologies. CASE tool support for (1) quick and easy development of new models and, in particular, (2) comfortable modification of existing models then appears as an indispensable prerequisite for cost-effectively handling their development. Tool-supported refactorings of behavior models, like the work presented in this paper, appear to be one step towards more comfortable and cheaper model-based development processes.

Future work is bound (1) to extended implementations of the satisfiability checker that is needed for the reduction of refactored transitions, (2) to the tight integration of our approach into a CASE tool that, in particular, must include the automatic layouting of computed STDs, and (3) to an extension to other formalisms, e.g., statecharts with OCL. While we believe that working with logical characterizations of control states is a viable option to refactoring state machines, we need more experience to identify situations where which model refactorings are of considerable methodological value, where not, and why.

Appendix A: Proof of equivalence

We show that the transformation w.r.t. a covering P, given in Eq. (1) in Sect. 4.4, is indeed a refactoring, i.e., $[\![R]\!] \overset{I \cup O}{=} [\![\tilde{R}]\!]$. We prove the stronger claim, $[\![R]\!] = [\![\tilde{R}]\!]$. Restrictions to the I/O behavior are necessary only if the set of local data state variables, L, is modified. We have moved modifications of this set – more precisely, of state variable *state* – into the mappings τ and τ^{-1} that transform rule systems into state machines, and vice versa. We need to show that for all pairs of subsequent states, $\beta\gamma$, of traces in R, there is a transition $\tilde{t}$ of $\tilde{R}$ with $\beta, \gamma' \models \tilde{t}$, and vice versa. Both directions are proved by induction.

"$\subseteq$". In order to show $[\![R]\!] \subseteq [\![\tilde{R}]\!]$, we first show that the first state of a trace of the former also is the first state of a trace of the latter. This follows directly because R and $\tilde{R}$ have the identical assertion S for initial states.

For the induction step, consider two subsequent states β and γ of a trace of R, i.e., $\ldots\beta\gamma\ldots \in [\![R]\!]$. By definition, there must be a transition $t \in T$ with $\beta, \gamma' \models t$ where $\beta, \gamma \in A_{V \cup H_t}$. Let $t \equiv in \wedge g \wedge a \wedge out$. We have to show that there are $p, q \in P$ with $\beta, \gamma' \models p \wedge q'[f_l/l']_{l \in L}$.

Since P covers the data space, A_L, there must be $p, q \in P$ s.t. $\beta \models p$ and $\gamma \models q$, or equivalently, $\gamma' \models q'$. By definition, $a \equiv \bigwedge_{l \in L} l' = f_l$, and because t implies a, it is the case that $\beta, \gamma' \models t$ implies $\beta, \gamma' \models \bigwedge_{l \in L} l' = f_l$. Hence $\beta, \gamma' \models p \wedge q' \wedge \bigwedge_{l \in L} l' = f_l$.

By definition, we have $q'[f_l/l']_{l \in L} \equiv q' \wedge \bigwedge_{l \in L} l' = f_l$. Consequently, $\beta, \gamma' \models p \wedge q'[f_l/l']_{l \in L}$. $\beta, \gamma' \models t$ implies $\beta, \gamma' \models in \wedge g \wedge out$. Altogether, this yields $\beta, \gamma' \models in \wedge g \wedge p \wedge q'[f_l/l']_{l \in L} \wedge a \wedge out$. This shows that if γ is reachable from an initial state β in R, then this is also the case in $\tilde{R}$.

"$\supseteq$". In order to show $[\![R]\!] \supseteq [\![\tilde{R}]\!]$, we already know that the first state of a trace of $\tilde{R}$ also is one of a trace of R. Consider subsequent states β and γ of a trace of $\tilde{R}$. There is a $\tilde{t} \in \tilde{T}$ with $\beta, \gamma' \models \tilde{t}$. By construction of $\tilde{T}$, there also is a $t \in T$ with $\tilde{t} \Rightarrow t$ and consequently, $\beta, \gamma' \models t$.

Appendix B: Proof of preservation of determinism

Proof A shows that the traces of a rule system and its refactored counterpart are equivalent. However, this does not prevent nondeterminism from being introduced, as the example in Sect. 4.6 shows (it is irrelevant that the example takes into account the explicit *state* variable). The reason is that in the first direction of the proof ("$\subseteq$"), we show that for each transition t taken in the context of R, there *must be* a transition $\tilde{t}$ in the context of $\tilde{R}$. Now, the example shows that there can be more than one, even though different transitions may entail the same traces. As it turns out, this kind of internal nondeterminism is not introduced into deterministic systems if partitionings (and not just coverings) are used. This is shown in the remainder of this appendix.

A rule system $R = (V, S, T)$ with variables $V = L \cup I \cup O$ is called *deterministic* iff

1. the initial state is unique, i.e.,

$$\forall \beta, \gamma \in A_V \bullet \beta \models S \wedge \gamma \models S \Rightarrow \beta \overset{L}{=} \gamma, \text{ and} \quad (6)$$

2. in each state $\beta \in A_V$, at most one transition $t \in T$ is enabled, i.e.,

$$\forall s, t \in T\ \forall \beta, \gamma_s, \gamma_t \in A_V \bullet$$
$$\beta, \gamma_s' \models s \wedge \beta, \gamma_t' \models t \Rightarrow s \Leftrightarrow t. \quad (7)$$

We now show that if the rule system $R = (V, S, T)$ is deterministic and transformed to $\tilde{R} = (V, S, \tilde{T})$ w.r.t. a partitioning predicate set P, then $\tilde{R}$ is deterministic. We prove this claim by showing that Eqs. (6) and (7) hold for the transformed rule system $\tilde{R}$.

Equation (6) vacuously holds because R and $\tilde{R}$ have the identical initial condition S. For Eq. (7) consider two transitions $\tilde{s}, \tilde{t} \in \tilde{T}$ and states $\beta, \gamma_{\tilde{s}}, \gamma_{\tilde{t}} \in A_V$ with

$$\beta, \gamma_{\tilde{s}}' \models \tilde{s} \text{ and } \beta, \gamma_{\tilde{t}}' \models \tilde{t}. \quad (8)$$

By construction of $\tilde{T}$, there are transitions $u, v \in T$ and predicates $p_1, q_1, p_2, q_2 \in P$ with

$$\tilde{s} \Leftrightarrow p_1 \wedge u \wedge q_1'[f_{l_u}/l']_{l \in L} \text{ and } \tilde{t} \Leftrightarrow p_2 \wedge v \wedge q_2'[f_{l_v}/l']_{l \in L}. \quad (9)$$

Obviously, we have $\tilde{s} \Rightarrow u$ and $\tilde{t} \Rightarrow v$. Hence $\beta, \gamma_{\tilde{s}}' \models u$ and $\beta, \gamma_{\tilde{t}}' \models v$. Because R is deterministic, it follows that $u \Leftrightarrow v$, which in turn ensures $\gamma_{\tilde{s}}' = \gamma_{\tilde{t}}'$ and $\left(f_{l_u}\right)_{l \in L} = \left(f_{l_v}\right)_{l \in L}$.

Using these equivalences and the definition of $\models$ with primed variables, Eqs. (8) and (9) rewrite into $\beta \models p_1 \wedge u \wedge p_2$. P is a partitioning, i.e. $(p_1 \not\Leftrightarrow p_2) \Rightarrow \neg(p_1 \wedge p_2)$, which entails $p_1 \Leftrightarrow p_2$. Analogously, $\gamma_{\tilde{s}}' \models q_1'[f_{l_u}/l']_{l \in L}$ and $\gamma_{\tilde{s}}' \models q_2'[f_{l_u}/l']_{l \in L}$ yield $q_1 \Leftrightarrow q_2$. Altogether, we have $\tilde{s} \Leftrightarrow \tilde{t}$. $\tilde{R}$ is deterministic.

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theor. Comput. Sci. **138**(1), 3–34 (1995)
2. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: a successful application of B in a large project. In: Proceedings of Formal Methods, vol. 1708 of Springer LNCS, pp. 369–387 (1999)
3. Basin, D., Kuruma, H., Takaragi, K., Wolff, B.: Verification of a Signature Architecture with HOL-Z. In: Proceedings of Formal Methods, vol. 3582 of Springer LNCS, pp. 269–285 (2005)
4. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11.11 standard case-study. SW Pract. Exp. **34**(10), 915–948 (2004)
5. Beine, M., Otterbach, R., Jungmann, M.: Development of safety-critical software using automatic code generation. In: Proceeding SAE World Congress (publication SP-1852: In-Vehicle Networks and Software, Electrical Wiring Harnesses, and Electronics and Systems Reliability) (2004)
6. Breitling, M., Philipps, J.: Step by step to histories. In: Proceedings of Algebraic Methodology And Software Technology, vol. 1816 of Springer LNCS, pp. 11–25 (2000)
7. Broy, M., Stølen, K.: Specification and Development of Interactive systems – Focus on Streams, Interfaces, and Refinement. Springer, Berlin (2001)
8. Chan, W., Anderson, R., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.: Model checking large software specifications. IEEE TSE **24**(7), 498–520 (1998)
9. Cheng, Y.-P.: Refactoring design models for inductive verification. In: Proceedings of International Symposium on Software Testing and Analysis, pp. 164–168 (2002)
10. Functional Logic Language Curry. Language Homepage: www.informatik.uni-kiel.de/~mh/curry/, (2006)
11. Correa, A., Werner, C.: Applying refactoring techniques to UML/OCL Models. In: Proceedings of 7th International Conference on the Unified Modeling Language, pp. 173–187 (2004)
12. Dajani-Brown, S., Cofer, D., Hartmann, G., Pratt, S.: Formal modeling and analysis of an avionics triplex sensor voter. In: Proceedings of 10th International SPIN Workshop, volume 2648 of Springer LNCS, pp. 34–48 (2003)
13. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
14. Eckrich, M., Schäuffele, J., Baumgartner, W.: New Steering System-BMW on the road to success with ASCET-SD, ES1000 and INCA. RealTimes, **1**, 20–21 (2001) en.etas-group.com/downloads/rt/rt_2001_01_20_en.pdf
15. Ferrari, A., Gaviani, G., Gentile, G., Stefano, M., Romagnoli, L., Beine, M.: Automatic code generation and platform based design methodology: an engine management system design case study. In: Proceedings of SAE World Congress (publication SP-924: Software/Hardware Systems) (2005)
16. Farchi, E., Hartman, A., Pinter, S.S.: Using a model-based test generator to test for standard conformance. IBM Syst. J. **41**(1), 89–110 (2002)
17. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison Wesley, Reading (1999)
18. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Proceedings of 9th International Conference on Computer Aided Verification, vol. 1254 of Springer LNCS, pp. 72–83 (1997)
19. Hanus, M.: The integration of functions into logic programming: from theory to practice. J. Logic Program. **19–20**, 583–628 (1994)
20. Heninger, K.: Specifying software requirements for complex systems: new techniques and their application. IEEE TSE SE- **6**(1), 2–13 (1980)
21. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated consistency checking of requirements specifications. ACM Trans. SW Eng. Methodol. **5**(3), 231–261 (1996)
22. Huber, F., Schätz, B., Einert, G.: Consistent Graphical Specification of Distributed Systems. In: Proceedings of Formal Methods Europe, vol 1313 of Springer LNCS, pp 122–141 (1997)
23. Jeffords, R., Heitmeyer, C.: Automatic Generation of State Invariants from Requirements Specifications. In: Proceedings of 6th International Symposium on Foundations of SW Engineering, pp. 56–69 (1998)

24. Janicki, R., Sekerinski, E.: Foundations of the Trace Assertion Method of Module Interface Specification. IEEE TSE **27**(7), 577–598 (2001)
25. Lamport, L.: TLA in pictures. IEEE TSE, **21**(9), 768–775 (1995)
26. Lynch, N., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of 6th annual ACM Symposium on Principles of Distributed Computing, pp. 137–151 (1987)
27. Lynch, N., Vaandrager, F.: Forward and backward simulations for timing-based systems. In REX workshop, vol. 600 of Springer LNCS, pp. 397–446 (1991)
28. Manna, Z. et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Department of Computer Science, Stanford University (1994)
29. Mens, T., Demeyer, S., Du Bois, B., Stenten, H., Van Gorp, P.: Refactoring: current research and future trends.. ENTCS **82**(3), 483–499 (2003)
30. MOST Cooperation. MOST Specification, Rev. 2.2. www.mostnet.de/downloads/Specifications/(2002)
31. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE TSE **30**(2), 126–139 (2004)
32. Parnas, D.: Tabular Representations of Relations. Technical Report CRL-260, Telecommunications Research Institute of Ontario, (1992)
33. Parnas, D., Madey, J.: Functional documents for computer systems. . Sci. Comput. Program. **1**(25), 41–61 (1995)
34. Parnas, D., Peters, D.: An easily extensible toolset for tabular mathematical expressions. In: Proceedings of 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 1579 of Springer LNCS, pp. 345–359 (1999)
35. Prowell, S., Poore, J.: Foundations of sequence-based software specification. IEEE TSE **29**(5), 1–13 (2003)
36. Pretschner, A., Prenninger, W.: Computing refactorings of behavior models. In: Proceedings of ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, vol. 3713 of Springer LNCS, pp. 126–141 (2005)
37. Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S., Scholl, K.: Model-based test case generation for smart cards. ENTCS **80**, 168–192 (2003)
38. Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Zölch, R., Sostawa, B., Stauner, T.: One evaluation of model-based testing and its automation. In: Proceedings of 27th International Conference on Software Engineering, pp. 392–401 (2005)
39. Philipps, J., Rumpe, B.: Refinement of information flow architectures. In: Proceedings of 1st International Conference on Formal Engineering Methods, pp. 203–212 (1997)
40. Philipps, J., Rumpe, B.: Refinement of pipe and filter architectures. In: Proc. of World Congres on Formal Methods, vol. 1708 of Springer LNCS, pp. 96–115 (1999)
41. Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S.: Model Based Testing for Real—The Inhouse Card Case Study. J Softw. Tools Technol. Transf. **5**(2–3), 140–157 (2004)
42. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.: Refactoring UML models. In: Proceedings of 4th International Conference on the Unified Modeling Language, vol. 2185 of Springer LNCS, pp. 134–148 (2001)
43. Shen, H., Zucker, J., Parnas, D.: Table transformation tools: Why and how. In: Proceedings 11th Annual Conf. on Computer Assurance, pp. 3–11 (1996)
44. van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards Automating Source-Consistent UML Refactorings. In: Proceedings 6th International Conference on The Unified Modeling Language, Modeling Languages and Applications, vol. 2863 of Springer LNCS, pp. 144–158 (2003)