Computing Refactorings of Behavior Models

Alexander Pretschner[1] and Wolfgang Prenninger[2]

[1] Information Security, ETH Zürich, 8092 Zürich, Switzerland
[2] BMW Group, 80788 München, Germany
`Alexander.Pretschner@inf.ethz.ch, Wolfgang.Prenninger@bmw.de`

Abstract. For given behavior models expressed in statechart-like formalisms, we show how to compute semantically equivalent but 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.

1 Introduction

The use of explicit models is enjoying an increasing popularity in the development of complex systems. Modeling languages, including UML, 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. The (behavior) models are then used to generate simulation and production code, code skeletons, or test cases. They are also subjected to formal verification technology such as model checking or automated deductive theorem proving. While there is no fit to all needs yet, the respective technology is impressive, and systems of considerable complexity can be handled.

The increasing complexity of these systems necessitates the study of the development of the models itself. The context of this paper is the incremental development of models. We study one particular development step in such processes: refactoring [1,2] denotes structural transformations of a system that do not change its externally visible behavior, except maybe for memory allocation or required processor cycles. Code-based examples 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 bubbles. 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 [3] that we used to model industry-size systems to the end of test case generation (e.g., [4,5,6]). 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 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: one control state with 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. 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 [7]. We do not discuss how to pick p and q. The approach is prototypically implemented.

We present our ideas on the grounds of the simple example of a stack. As a proof of concept, we show how our techniques have been used in the case study of an automotive network controller [4]. 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 like those in SCR [8,9,10]. 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 [11,10]. 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 [12,13].

To summarize, we tackle the following *problem*. In the context of incremental development, assume a state machine, or a table, and a partitioning of the state space, to be given. How can we compute an equivalent state machine with a set of control states characterized by a set of 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 behavior models on the grounds of partitions of the state space. Our approach generalizes to other formalisms as well. Statecharts, for instance, may in principle arbitrarily access the 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

Section 2 presents the formalism of this paper 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, given by both tables and state transition diagrams. Given a partitioning of the state space, Section 4 shows how to compute refactorings and briefly considers the implementation. Section 5 presents the application of the approach in an industrial case study. Sections 6 and 7 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. *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.

Preliminaries The formalism borrows from Breitling and Philipps [14]. 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 V. Let $\mathit{free}(\Phi)$ denote the set of free variables in a logical formula Φ. In case an assertion Φ evaluates to true when all $v \in \mathit{free}(\Phi)$ are replaced by $\beta(v)$, we write $\beta \models \Phi$.

Variable names also occur in primed form (intuition given in the next paragraph 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 $\mathit{free}(\Phi)$. 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 $\mathit{free}(\Phi)$ 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, $Y_1 \overset{W}{=} Y_2$ denotes $\forall y_1 \in Y_1 \exists y_2 \in Y_2 \bullet y_1 \overset{W}{=} y_2$ 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—the names of the functions defined in the action language and used in guards and assignments. The type of a term t is denoted by $\mathit{type}(t)$. 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*.

Rule Systems A *rule system* is a tuple $R = (V, S, T)$. V consists of disjoint sets of typed variables, I, O, L. They denote input, output, and local variables, respectively. A *state* of R is a valuation $\beta \in A_V$ that type-correctly maps all variables in V to ground terms. $\beta \in A_L$ is called a *data state* of R.

S is an assertion with $\mathit{free}(S) \subseteq V$. 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 $\mathit{free}(t) \subseteq V \cup V'$. 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 in T to be of the form $\mathit{in} \wedge g \wedge a \wedge \mathit{out}$. 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 $\mathit{type}(\pi_i) = \mathit{type}(i)$. 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 $H_t \cup L$, with $\mathit{type}(g) = \mathit{Bool}$. 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 $\mathit{type}(f_l) = \mathit{type}(l)$. Finally, $\mathit{out} \equiv \bigwedge_{o \in O} o' = f_o$ 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 $\mathit{type}(f_o) = \mathit{type}(o)$. ε denotes the absence of signals both for input and output channels; 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 *run* 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 runs, 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 run, 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 [15], input enabledness [16], fairness, etc. Rule systems need not be total nor deterministic.

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 $\mathit{state} = \overline{\mathit{src}}$, and each assignment to contain a statement $\mathit{state}' = \overline{\mathit{dst}}$ where $\overline{\mathit{src}}$ and $\overline{\mathit{dst}}$ are the 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)—bubbles (control states) and arrows (transitions). Two examples of (incomplete) STDs are given in Fig. 1. The black dot denotes the initial state.

Every state machine is a rule system, but not each 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 $\{\overline{s}\}$ to L, add the conjunct $\mathit{state} = \overline{s}$ to the guard of each transition, and add the conjunct

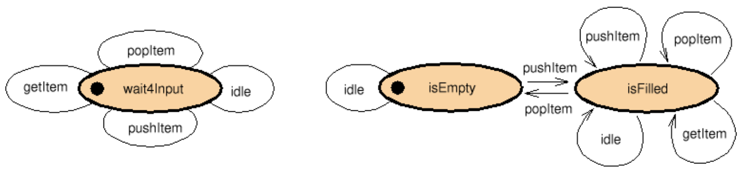

Fig. 1. Original STD of the stack (left); refactoring (right)

$state' = \overline{s}$ 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 [13]. For us, any tabular representation will do. An example of a table is given in Tab. 1.

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

Table 1. Behavior of a stack

Example Consider the specification of a stack of integers. We assume a component with one input channel $I = \{e\}$ with $type(e) = \{push(Int), get, pop, \varepsilon\}$, and one output channel, $O = \{a\}$ 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 `rt(list(X,Y)) = Y`. One transition-local variable is used in the example, namely `DATA` in transition *pushItem*.

By adding a further local variable *state* of $type(state) = \{\overline{wait4Input}\}$ to the set L of local variables, we generate a state machine from the rule system by also adding trivial statements $state = \overline{wait4Input}$ and $state' = \overline{wait4Input}$ to guard and assignment of each row of Tab. 1. Fig. 1, left, 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 on these steps (except for enforceable consistency conditions that we do not discuss here).

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 useful 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 modifications in STDs are rather tedious: because the control states of the state machine change, transitions or parts of transitions have to be copied or removed multiple times. This is an error-prone and tedious task.

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 (Section 5).

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 [15,17,18]. *Interface modifications* add or delete input or output channels to or from a system. If, before deletion, the name of a channel does not occur in a system's description, its removal does not change the system's behavior, and neither does the introduction of a new channel. *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 extensions 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: refactorings of state machines ($\rho_S \in \{\rho | [\![R]\!] \overset{I \cup O}{=} [\![\rho(R)]\!]$ *and* R *is a state machine*$\}$), refactorings of rule systems ($\rho_R \in \{\rho | [\![R]\!] \overset{I \cup O}{=} [\![\rho(R)]\!]$ *and* R *is a rule system*$\}$), and modifications of rule systems and state machines (both denoted by δ in Fig. 2). Modifications modify, add, or delete transitions, possibly with alterations of L.

Let τ and τ^{-1} denote transformations from rule systems into state machines, and vice versa. Fig. 2 illustrates the relationship between the development steps. As development progresses from top to bottom, modifications take place. Within each row, usually different refactorings of both tables and state machines are considered, and the further can be transformed into the latter, and vice versa.

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. However, for reasons that we will be able to explain only after refactorings have been made precise, it is not always desirable to let $\tau^{-1} = id$.

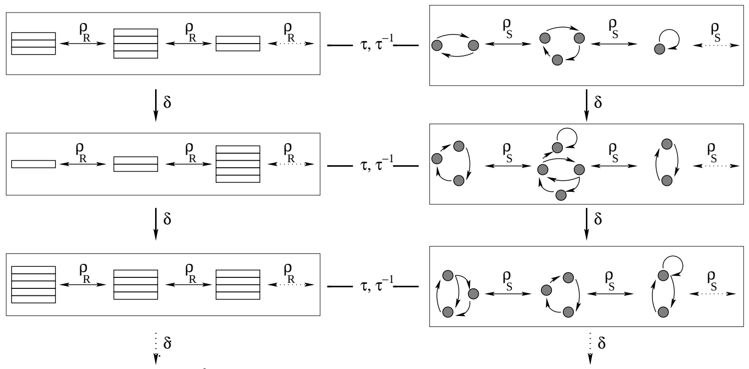

Fig. 2. Incremental Development

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 Section 1. Methodologically, one 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

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 paper, the idea of refactoring state machines or rule systems is to define a set of predicates that partition the data space. In general, whether a set of predicates forms a partitioning is undecidable. In our case studies, however, we could easily see whether or not a set of predicates formed a partitioning. Each of these 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 partitioning predicates have been defined, one must compute the transitions between the corresponding states.

To get an intuition of this computation, assume a set of predicates, P, that partition 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

Name	Guard	Input	Output	Assignment
~~*pushItem*~~	~~*isE(st) ∧ isE(list(DATA,st))*~~	~~*e≅push(DATA)*~~		~~*st'=list(DATA,st)*~~
pushItem	isE(st) ∧ not(isE(list(DATA,st)))	e≅push(DATA)		st'=list(DATA,st)
~~pushItem~~	~~not(isE(st)) ∧ isE(list(DATA,st))~~	~~e≅push(DATA)~~		~~st'=list(DATA,st)~~
pushItem	not(isE(st)) ∧ not(isE(list(DATA,st)))	e≅push(DATA)		st'=list(DATA,st)
~~getItem~~	~~not(isE(st)) ∧ isE(st) ∧ isE(st)~~	~~e≅get~~	~~a'=ft(st)~~	
~~getItem~~	~~not(isE(st)) ∧ isE(st) ∧ not(isE(st))~~	~~e≅get~~	~~a'=ft(st)~~	
~~getItem~~	~~not(isE(st)) ∧ not(isE(st) ∧ isE(st))~~	~~e≅get~~	~~a'=ft(st)~~	
getItem	not(isE(st)) ∧ not(isE(st)) ∧ not(isE(st))	e≅get	a'=ft(st)	
~~popItem~~	~~not(isE(st)) ∧ isE(st) ∧ isE(rt(st))~~	~~e≅pop~~		~~st'=rt(st)~~
~~*popItem*~~	~~*not(isE(st)) ∧ isE(st) ∧ not(isE(rt(st)))*~~	~~*e≅pop*~~		~~*st'=rt(st)*~~
popItem	not(isE(st)) ∧ not(isE(st)) ∧ isE(rt(st))	e≅pop		st'=rt(st)
popItem	not(isE(st)) ∧ not(isE(st)) ∧ not(isE(rt(st)))	e≅pop		st'=rt(st)
idle	isE(st) ∧ isE(st)	e≅ε		
~~idle~~	~~isE(st) ∧ not(isE(st))~~	~~e≅ε~~		
~~idle~~	~~not(isE(st)) ∧ isE(st)~~	~~e≅ε~~		
idle	not(isE(st)) ∧ not(isE(st))	e≅ε		

Table 2. Refactored behavior

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.

Example Consider the stack again. 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. Tab. 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, 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 line of Tab. 2 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 which is why this transition is also canceled out.

Fig. 1, right, shows the STD of the stack as defined by Tab. 2 that we assume to be extended by the respective assignments to *state* and *state'*. Transitions are abbreviated. *isFilled* denotes the control state that is defined by $not(isE(st))$.

Formalization We will now make the refactoring step precise. Let P denote a finite set of predicates that partition the data space of a rule system $R = (V, S, T)$ with $V = I \cup O \cup L$ defined as above. The partitioning requirement means firstly that P covers A_L, i.e. for all states β, we have $\beta \models \bigvee_{p \in P} p$. Secondly, the predicates in P must be pairwise disjoint, i.e. $\forall p, q \in P \bullet p \neq q \Rightarrow \neg(p \wedge q)$. For convenience, we also require that all predicates in P be satisfiable and an initial partition be uniquely defined, i.e. $\exists s \in P \bullet S \Rightarrow s$ because of the partitioning requirement. Refactoring a rule system $R = (V, S, T)$ w.r.t. a partitioning P of the data space yields a rule system $\rho_R(R) = \tilde{R} = (V, S, \tilde{T})$ with

$$\tilde{T} := \Big\{ in \ \wedge \ g \wedge p \wedge q'[f_l/l']_{l \in L} \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o \mid \\ (in \ \wedge \ g \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o) \in T \ \wedge \{p, q\} \subseteq P \Big\}.$$

The proof that the transformation is indeed a refactoring, i.e. $[\![R]\!] \overset{I \cup O}{=} [\![\tilde{R}]\!]$, is given in Appendix A. The proof only requires P to cover the state space; partitioning ensures that no internal nondeterminism is introduced.

If one wants to perform the refactoring and generate a state machine in one step (Section 3), then the following construction can be used. With a new variable $state \in \tilde{L}$ 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

$$\tilde{T} := \Big\{ in \ \wedge \ g \wedge p \wedge q'[f_l/l']_{l \in L} \ \wedge \ state = \overline{p} \ \wedge \ state' = \overline{q} \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \\ \bigwedge_{o \in O} o' = f_o \mid (in \ \wedge \ g \ \wedge \ \bigwedge_{l \in L} l' = f_l \ \wedge \ \bigwedge_{o \in O} o' = f_o) \in T \ \wedge \ \{p, q\} \subseteq P \Big\}.$$

Removing state variables 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 is likely to clutter the model (more precisely, guards and assignments of transitions). This is the only reason for not letting $\tau^{-1} = id$ (Section 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.

We will now characterize the operations that, upon application of τ^{-1} allow one to delete *state* variables in rule systems that were previously introduced by the application of τ. As explained in Section 3, it is sufficient to focus on behavior modifications, and to ignore interface modifications.

The above construction of computing a refactoring and a state machine in one step shows that $state = \overline{p}$ whenever p holds. Conversely, we have $state' = \overline{q}$ whenever $q'[f_l/l']_{l \in L}$ evaluates to true. In other words, the information on the explicit state variable is indeed redundant and can be removed (it is only used to decide whether or not to draw a transition arrow between two control states).

The same is true for modifications of existing transitions (including modifications of the data state $L - \{state\}$), and also for the deletion of transitions. New transitions between control states that are characterized by $p, q \in P$ are equally unproblematic if some implementing CASE tool adds $p \wedge q'[f_l/l']_{l \in L}$ to the guard of a transition from $\overline{p}$ to $\overline{q}$ (by removing assignments to *state* and *state'*; this is the—informal—definition of τ^{-1}). The only problem occurs if a new control state plus transitions to or from it are added at the graphical level *without* giving a logical characterization of this control state. This is problematic because in this case, it is not possible to automatically modify guards and assignments as in the case of logically characterized control states.

In other words, if the CASE tool forbids the introduction of new control states at the *graphical level* when no logical characterization is provided and, instead, requires development steps of this kind to be performed at the *level of tables* only, then we can work with tables and STDs in parallel, without cluttering the model. In this case, refactorings of state machines are computed via $\rho_S = \tau \circ \rho_R \circ \tau^{-1}$ rather than via ρ_R.

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 simplification, possibly to *false*. Step (b) is particularly important because the computed transitions should be readable by humans, and, as the examples of this paper show, there is a great potential for the removal of redundant parts. Our simplification algorithm implements the rules of Boolean algebra and includes a simple satisfiability checker. The latter is used to remove unsatisfiable disjuncts for formulas in disjunctive normal form. The 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—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 [19] (the operational semantics of which relies on narrowing [20] which explains why it lends itself to satisfiability checking). With a restriction of all lists to a maximum length of 5, the example in the next section is computed in negligible time. We have not yet implemented a plugin that also takes into account automatic layouting of computed STDs.

5 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) [21]. The model was the basis for model-based testing of an NM implementation [4]. The functionality of the network is divided into function blocks which reside on the network's devices. The NM is a special function block responsible for network management. Here 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.

We do not show any complex modeling details here and describe only the main local variables of the model. The model defines the variable *mode* which models the five modes of the NM: 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. Furthermore the model defines the variable *wa* which stores the network address from which the NM expects an answer to its last request. There are four additional variables for storing the central registry and other informations about the system.

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. We transformed this table into different state machines for a review of the model. We choose the partitioning P_1 which divides the state space according to the five modes of the NM. Fig. 3, left, depicts the respective state machine. In addition, we choose a second partitioning P_2 which distinguishes between states (1) $\mathit{requestingDevices} \equiv \mathit{wa} = \mathit{empty} \wedge \mathit{mode} \in \{\mathit{init}, \mathit{ncd}, \mathit{delayed}\}$ where the NM requests devices, (2) $\mathit{waitForStatus} \equiv \mathit{wa} \neq \mathit{empty} \wedge \mathit{mode} \in \{\mathit{init}, \mathit{ncd}, \mathit{delayed}\}$ where the NM waits for an answer, and the states (3) *off* and (4) *cfgOk* where the NM is in modes *off* or *cfgOk*. Fig. 3, right, depicts the state machine w.r.t. partitioning P_2.

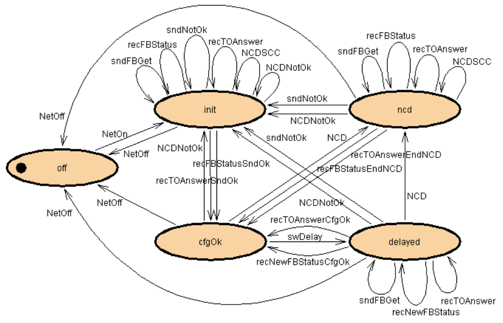

Fig. 3. STD of the NM w.r.t. partitioning P_1 (left); w.r.t. partitioning P_2 (right)

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 reveals that specific symmetries can be found and analyzed by building different abstract views of behavior models. 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 have stayed hidden in the detailed view of tables.

6 Related work

Refactorings: Sunyé et al. consider the refactoring of statecharts on the grounds of hierarchical states [22]. 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). In the context of inductive verification, Cheng considers refactoring a parameterized process into a set of constant processes [23]. In our context, this would amount to refactoring one state machine into more than one state machine. Van Gorp et al. propose extensions to the UML meta model such that pre- and postconditions for behavior-preserving transformations can be expressed [24]. This work is not concerned with refactorings of state machines. In a similar vein, Correa and Werner discuss refactorings of OCL expressions and class structures, without explicitly taking into account state machines [25]. Philipps and Rumpe present a set of transformation rules for data flow networks and formally show that the transformed system is a refinement of the original one [18]. Their work differs from ours in that we actually compute the refactoring of a behavior model.

Tables and Incrementality: Shen et al. [12] are concerned with transformations of tabular specifications of a system. They concentrate on transformations between different kinds of tables [13] rather than transforming tables into graphical representations in the form of extended state machines. Their transformations are refactorings in their own right. Prowell and Poore use incrementally discovered equivalence classes on I/O sequences to specify the I/O behavior of a system [26]. One could directly use such canonical sequences as states. Janicki and Sekerinski claim that this leads to complex state machines even for small systems [27]. In that 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 [28,29] are directly 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 [7] 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 that we use to characterize control states relate to the "reaffirmed invariants" in the context of STeP [30], 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 [31].

7 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" [1, p. 75].

We have shown a way of computing refactorings of state machines on the grounds of predicates that describe parts of the state space: local invariants. 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 beneficiary to use them in parallel: because of their clear structure, tables are sometimes easier to grasp—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 singled out one particular development step—the introduction of transitions from or to control states with no logical characterization—that should be performed at the level of tables rather than state machines.

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. Refactorings of behavior models, like the work presented in this paper, are 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.

Acknowledgments J. Philipps pointed us to Lamport's work on predicate-action diagrams. B. Schätz and B. Seybold provided useful comments on this paper.

References

1. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley (1999)
2. Mens, T., Demeyer, S., Du Bois, B., Stenten, H., Van Gorp, P.: Refactoring: Current Research and Future Trends. In: Proc. ETAPS 2003 Workshop on Language Descriptions, Tools and Applications. (2003)
3. Huber, F., Schätz, B., Einert, G.: Consistent Graphical Specification of Distributed Systems. In: Proc. Formal Methods Europe. (1997) 122 – 141
4. 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: Proc. 27th Intl. Conf. on Software Engineering. (2005) 392–401
5. Philipps, J., Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S., Scholl, K.: Model-based test case generation for smart cards. In: Proc. 8th Intl. Workshop on Formal Methods for Industrial Critical Systems. (2003) 168–192
6. Pretschner, A., Slotosch, O., Aiglstorfer, E., Kriebel, S.: Model Based Testing for Real—The Inhouse Card Case Study. J. STTT **5** (2004) 140–157
7. Lamport, L.: TLA in Pictures. IEEE TSE **21** (1995) 768–775
8. Heninger, K.: Specifying Software Requirements for Complex Systems: New Techniques and Their Application. IEEE TSE **SE-6** (1980) 2–13
9. Parnas, D., Madey, J.: Functional Documents for Computer Systems. Science of Computer Programming **1** (1995) 41–61
10. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated Consistency Checking of Requirements Specifications. ACM Trans. on SW Eng. and Meth. **5** (1996) 231–261
11. Parnas, D., Peters, D.: An Easily Extensible Toolset for Tabular Mathematical Expressions. In: Proc. TACAS'99. (1999) 345–359
12. Shen, H., Zucker, J., Parnas, D.: Table transformation tools: Why and how. In: Proc. 11th Annual Conf. on Computer Assurance. (1996) 3–11
13. Parnas, D.: Tabular Representations of Relations. Technical Report CRL-260, Telecommunications Research Institute of Ontario (1992)
14. Breitling, M., Philipps, J.: Step by step to histories. In: Proc. Algebraic Methodology And Software Technology. Volume 1816 of Springer LNCS. (2000) 11–25
15. Broy, M., Stølen, K.: Specification and Development of Interactive Systems – Focus on Streams, Interfaces, and Refinement. Springer (2001)
16. Lynch, N., Tuttle, M.: Hierarchical correctness proofs for distributed algorithms. In: Proc. 6th annual ACM symp. on principles of distr. computing. (1987) 137–151
17. Philipps, J., Rumpe, B.: Refinement of information flow architectures. In: Proc. ICFEM'97. (1997)
18. Philipps, J., Rumpe, B.: Refinement of pipe and filter architectures. In: FM'99, LNCS 1708. (1999) 96–115
19. Hanus, M.: Functional Logic Language Curry. Language Hompage: http://www.informatik.uni-kiel.de/~mh/curry/ (2005)
20. Hanus, M.: The integration of functions into logic programming: From theory to practice. J. Logic Programming **19,20** (1994) 583–628
21. MOST Cooperation: MOST Specification, Rev. 2.2. http://www.mostnet.de/downloads/Specifications/ (2002)
22. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.M.: Refactoring UML models. In: Proc. 4th Intl. Conf. on the Unified Modeling Language. (2001) 134–148
23. Cheng, Y.P.: Refactoring design models for inductive verification. In: Proc. Intl. Symp. on Software Testing and Analysis. (2002) 164–168

24. van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards Automating Source-Consistent UML Refactorings. In: Proc. UML. (2003) 144–158
25. Correa, A., Werner, C.: Applying Refactoring Techniques to UML/OCL Models. In: Proc. 7th Intl. Conf. on the Unified Modeling Language. (2004) 173–187
26. Prowell, S., Poore, J.: Foundations of Sequence-Based Software Specification. IEEE TSE **29** (2003) 1–13
27. Janicki, R., Sekerinski, E.: Foundations of the Trace Assertion Method of Module Interface Specification. IEEE TSE **27** (2001) 577–598
28. Lynch, N., Vaandrager, F.: Forward and backward simulations for timing-based systems. Volume 600 of Springer LNCS. (1991) 397–446
29. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science **138** (1995) 3–34
30. Manna et al., Z.: STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Dept. of Computer Science, Stanford University (1994)
31. Jeffords, R., Heitmeyer, C.: Automatic Generation of State Invariants from Req. Specifications. In: Proc. 6th Intl. Symp. on Foundations of SW Engineering. (1998)

A Proof

We show that the given transformation w.r.t. a partitioning P 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 runs 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 run of the further also is the first state of a run 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 run 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 partitions 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 run of $\tilde{R}$ also is one of a run of R. Consider subsequent states β, γ of a run 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, $\beta, \gamma' \models t$.