A Unified Formulation of Behavioral Semantics for SysML Models

Christoph Hilken[1] Jan Peleska[1] Robert Wille[1,2]
[1]*Institute of Computer Science, University of Bremen, 28359 Bremen, Germany*
[2]*Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany*
{*chilken,jp,rwille*}*@informatik.uni-bremen.de*

Keywords: SysML, State Machines, Transition Relation, Model Checking, Model-based Testing

Abstract: In order to cope with the complexity of today's system designs, higher levels of abstraction are considered. Modeling languages such as SysML provide adequate description means for an abstract specification of the structure and the behavior of a system to be implemented. Due to its sufficient degree of formality, SysML additionally allows for performing several automated test and verification tasks. For these tasks, however, a formal encoding of the behavioral model semantics is required; this is typically achieved by generating initial state conditions as well as the transition relation from the model. Since SysML provides a multitude of alternative or complementary notations, this poses a significant challenge to the development of corresponding tool support. In this paper, we therefore propose an alternative approach to the generation of transition relations: In a first step, a model-to-model transformation is applied which unifies the behavioral descriptions into one single notation, namely operations allocated in blocks and specified by pre- and post-conditions. Afterwards, only pre- and post-conditions as well as some auxiliary constraints for fixing semantic variation points need to be considered when generating the transition relation. The approach presented here has been evaluated in the development of industrial tools supporting bounded model checking and model-based test generation.

1 INTRODUCTION

The design of systems including hardware as well as software components has become a cumbersome task. An increasing number of components, a tighter hardware/software interaction, or the integration of cyber-physical components such as sensors and actuators have led to a significant complexity to be tackled.

In the early phases of the design modeling languages such as UML (Object Management Group, 2011a) and its profiles SysML (Object Management Group, 2010) and MARTE (Object Management Group, 2011b), as well as others have shown to be beneficial. SysML, for example, provides several description means such as block definition diagrams (for the structure of a system), activity diagrams (for the behavior of operations in a system), or state machines (for control states of a system and their transitions) to precisely specify the structure and behavior of a system prior to its implementation. Constraints provided in OCL (Object Management Group, 2012) allow to additionally refine the respective descriptions with requirements to be satisfied. While this can always be used as a blueprint for the implementation phase, the corresponding (formal) descriptions additionally allow for performing several automated test and verification tasks, even if an implementation is not available yet.

As a consequence, several methods and approaches have been presented in the recent past utilizing these formal descriptions, e.g. (1) to prove that the specification is free of contradictory requirements (see e.g. (Gogolla et al., 2009)), (2) to check for unwanted behavior to be avoided (see e.g. (Soeken et al., 2011; Hilken et al., 2014)), or (3) to generate tests for the actual implementation (see e.g. (Peleska, 2013)). Indeed, very powerful and complementary solutions have been investigated, including approaches based on theorem provers such as Isabelle (Brucker and Wolff, 2006) or the KeY approach (Beckert et al., 2007) as well as based on automatic solving engines such as CSP solvers (Cabot et al., 2008) or SMT solvers (Soeken et al., 2011).

These methods, however, usually rely on a translation of the model under consideration to its corresponding transition relation, for the purpose of encoding the model's behavioral semantics. This is a laborious and, at the same time, highly critical task in which any error will significantly spoil the verification or validation result. Moreover, modeling formalisms like SysML provide a *multitude* of alternative or complementary notations for behavioral descriptions (i.e. state machines, activities, interactions, use cases, and operations) and existing approaches need to be provided with a transition relation for each and every one of those. This poses a significant challenge to the development of automated approaches for the verification and validation of models.

In this work, we aim for addressing this issue by proposing a two-step approach for the generation of the desired transition relation. First, a syntactic model-to-model transformation of the given behavioral descriptions into newly added operations specified by pre- and post-conditions and allocated in blocks is performed. This unifies the description means of a system's structure and behavior and, through this unification, simplifies the generation of the transition relation to be conducted in the second step. As a result, support for further behavioral descriptions can be added more easily into approaches for verification and test.

The effectiveness of the proposed approach is exemplified by translating behavioral state machine descriptions into blocks and operations, and creating the transition relation from the latter representation. The approach has been evaluated during the development of an industrial-strength tool for bounded model checking and model-based testing with SysML models.

The remainder of this paper is structured as follows: The next section briefly reviews the basic concepts of SysML while Section 3 sketches the problem formulation as well as the general idea of our solution. In Section 4 the transformation of state machines (serving as an example of a formalism for which a transition relation generator is required) into block diagrams and associated operations is described. This section is complemented by Section 5, where the transformation of blocks/operations into the propositional representation of transition relations is explained. In Section 6, we report on an actual implementation following the strategy described in this paper. We conclude with a summary in Section 7.

2 BACKGROUND

The *OMG Systems Modeling Language SysML* (Object Management Group, 2010) offers description means to specify the structure and the behavior of a system. The structure of a system can be described by means of *Block Definition Diagrams* (BDD) consisting of blocks and associations/relations between blocks. A block has different features, which are grouped in compartments, such as *properties* or *operations*. The former represent the state of the component represented by the block, the latter its behavior. Using design-by-contract, the effect of operations can be expressed by means of pre- and post-conditions. To this end, the *Object Constraint Language* (OCL) (Object Management Group, 2012) can be used. The SysML also offers descriptions means to explicitly specify the behavior of components; examples are activity diagrams, sequence diagrams, or state machines. For illustrating the objectives of this paper, state machines are used.

When referring to the elements of SysML state machines, some auxiliary functions are needed. In tool implementations, these functions are typically realized as operations defined on the abstract syntax tree used for internal model representation. Given a SysML state machine *sm* and a transition t^{sm}, the source state of this transition is denoted by $t^{sm}.source$, and its target state by $t^{sm}.target$. The action associated with the transition arrow is denoted by $t^{sm}.effect$. The entry action associated with a state machine state s is denoted by $s.entry$, the exit action by $s.exit$, and a do activity by $s.do$. The set of transitions emanating from s is denoted by $s.outgoing$. For dealing with *composite states* s in hierarchic state machines, function $parent(s)$ returns the immediate parent state of s. The whole state machine *sm* is considered as the root composite state of a hierarchic state machine, so $parent(s) = sm$, if s resides on the highest level of a state hierarchy. Function $subs(s)$ returns the set of all true sub-states of s, regardless of their position in the state machine hierarchy. For *simple* states s, $subs(s) = \varnothing$. Composite states s are characterized by $subs(s) \neq \varnothing$; function $sub(s)$ returns the pseudo initial state of the composite state's immediate submachine. $sub(sm)$ is the pseudo initial state of the whole state machine. If s_0 is a pseudo initial state, then $s_0.t$ denotes its single outgoing transition.

Example 1. *For the hierarchic state machine depicted in Figure 3(a),* $sub(sm)$ *is represented by the black bullet point with transition* $sub(sm).t$ *connecting the pseudo initial state to* s_0*. Action* $s_3.entry$ *consists of an assignment* $z = z+1$*;* $s_1.exit$ *of the assignment* $x = 1$*. The effect* $t3.effect$ *of transition* $t3$ *is the assignment* $r = 0$*.*

In this paper, we only consider single-region composite states, i.e., each composite state is associated with a sequential submachine (Object Management Group, 2011a, 15.3.11)[1]. Composite states induce *state configurations*, i.e., sets of hierarchically related states. The *active* configuration is the one the state machine currently resides in. Since we only consider decompositions into single regions, the active configuration of state machine *sm* is already fully defined by its innermost active simple state $state^{sm}$ and its ancestors $parent(state^{sm}), parent(parent(state^{sm})), \ldots, sm$.
The transitions emanating from hierarchic states are prioritized (Object Management Group, 2011a, p. 576): the lower the hierarchy level of a transition, the higher its priority.

Let $state^{sm}$ denote the innermost simple state the machine *sm* currently resides in. A transition t between a source state $t.source$ and a target state $t.target$ will be taken, if (1) $state^{sm}$ is equal to, or a substate of $t.source$, (2) the transition's guard $t.guard$ evaluates to true, and (3) no transition with higher priority could be taken. To specify the effect of transition t being taken, the *least common ancestor* $LCA(t.source, t.target)$ of source and target state has to be calculated (Object Management Group, 2011a, p. 584). This is the lowest composite state containing both $t.source$ and $t.target$.

Example 2. *Consider again the state machine sm from Figure 3(a). There* $LCA(s_2, s_3) = s_1$ *and* $LCA(s_0, s_3) = sm$ *holds.*

The *overall effect* of executing t is now specified as follows. (1) All exit actions, starting with $state^{sm}$, following the parent relation, and ending at the immediate substate of $LCA(t.source, t.target)$ are executed in that order. (2) The action associated with

[1]Composite states with multiple regions can be represented in an alternative way by constructing a separate state machine for each region and associating it with a different block; this alternative is covered by the approach described here.

the transition arrow, *t.effect*, is executed. (3) All entry actions, starting with the ancestor of *t.target* directly underneath $LCA(t.source, t.target)$, and ending at *t.target* are executed. (4) Descending from *t.target* until the simple target state – the new value of $state^{sm}$ – is reached, the action $s_0.t.effect$ associated with the unique outgoing transition t of pseudo initial state s_0 of each submachine is executed, and the entry action $s_0.t.target.entry$ associated with the target state reached from s_0 is performed. The execution of do activities is started after entering a state. Each do activity either terminates by itself while the state machine resides in the respective state, or it will be stopped as soon as this state is left.

3 PROBLEM FORMULATION AND GENERAL IDEA

Description means as described above allow for a very precise specification of a system to be realized. The resulting formal descriptions can already be utilized to perform automated test and verification tasks. For this purpose, a variety of (automated) methods and approaches have been developed in the recent past (see e.g. (Brucker and Wolff, 2006; Beckert et al., 2007; Cabot et al., 2008; Soeken et al., 2011; Hilken et al., 2014)).

A typical step involved in most of these approaches consists in transforming the model M or a part thereof into its transition relation Φ_M. The transition relation relates model states to their potential post-states. Even for infinite state models, the transition relation may be represented as a finite object by choosing its propositional representation where $\Phi_M(\sigma, \sigma')$ is a predicate relating pre-states σ to post-states σ'. In model-based testing and bounded model checking, for example, this representation is exploited by solving so-called *bounded model checking instances*, i.e.

$$b \equiv I(\sigma_0) \wedge \bigwedge_{i=1}^{k} \Phi_M(\sigma_{i-1}, \sigma_i) \wedge G(\sigma_0, \ldots, \sigma_k) \quad (1)$$

This constraint specifies a set of suitable pre-states σ_0 by proposition $I(\sigma_0)$. From such a starting state, the goal $G(\sigma_0, \ldots, \sigma_k)$ should be fulfilled when traversing the model for k steps. The conjunction $\bigwedge_{i=1}^{k} \Phi_M(\sigma_{i-1}, \sigma_i)$ requires that any solution of G shall be a legal trace through the model: any two consecutive states (σ_{i-1}, σ_i) in a solution of G have to be related by the transition relation. For bounded model checking applications, G typically denotes an *unwanted* property (e.g. a safety violation), so that a solution of b uncovers a modeling error (Biere et al., 2003). In model-based testing, however, b specifies a symbolic test case, and the solution of this constraint solving problem yields a suitable sequence of inputs to the system under test for covering the *test objective* G (Peleska, 2013).

However, following this scheme requires a transformation from the originally given model M to the transition relation based on the predicates $\Phi_M(\sigma, \sigma')$[2]. This is a laborious and, at the same time, highly critical task. Any error in this transformation will lead, for example, to invalid model checking results and to test data that is not really suitable for the objective to be covered. Moreover, modeling formalisms like SysML do not come with just a single notation for expressing behavior, but with a *multitude* of alternative or complementary notations: state machines (Object Management Group, 2010, Section 13), activities (Object Management Group, 2010, Section 11), interactions (Object Management Group, 2010, Section 12) (e.g. sequence diagrams), use cases (Object Management Group, 2010, Section 14), and operations (Object Management Group, 2010, Section 8) allocated in blocks. As a consequence, the corresponding model-to-text transformations usually have to be developed for each and every of these notations.

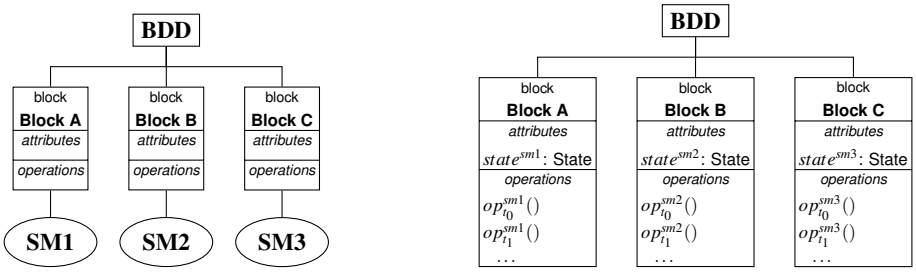
In this work, we present an effective approach which automates those transformations and, at the same time, makes its validation more reliable. To this end, we propose a two-step approach:

- First, a syntactic model-to-model transformation of the formalism under consideration into blocks and associated operations is performed. The behavior is then represented in terms of newly added operations which, in turn, are specified in OCL (Object Management Group, 2012) by means of pre- and post-conditions.
- Then, a semantic model-to-text transformation from blocks/operations into the transition relation is performed.

Example 3. *Fig. 1 illustrates the proposed approach by means of state machines. Fig. 1(a) provides the original model in which behavior is provided in terms of state machines. Those are translated into corresponding operations as shown in Fig. 1(b) which build the basis for the following translation into a transition relation.*

Following this scheme, the model-to-model transformation (first step) unifies the description means of a system's behavior and eventually leads to a model in which all operations are solely specified by pre- and post-conditions. Because of this, the translation into a transition relation only relies on a single notation (namely pre- and post-conditions) rather than multitude descriptions of behavior. Additional notations such as activities, interactions, etc. just have to be syntactically transformed into blocks and associated operations. Furthermore, many semantic variation points (Object Management Group, 2011a) can be implemented in the model-to-text transformation as well, so that these can also be re-used by means of alternative "compile switches" applied to the transformation from blocks/operations to $\Phi_M(\sigma, \sigma')$. We illustrate this by showing alternative instantiations

[2]Note that we consider this a *model-to-text transition* in the following.



(a) Source structure (b) Target structure

Figure 1: Proposed model-to-model transformation.

of the concurrency semantics (interleaving or synchronous).

In the following, both steps are described in detail by means of state machines[3].

4 MODEL-TO-MODEL TRANSFORMATION

In this section the model-to-model transformation is described. First, the generation of the target structure is introduced. Afterwards, the transformation of state machines consisting of simple states only is specified. This is eventually extended further to the transformation of composite states.

4.1 Generation of the Target Structure

The first step of the transformation is the generation of the target structure. For this purpose, properties and operations are added to the structure in order to represent the state machines' behavior. More precisely, given a state machine *sm* describing the behavior of a block b, the following properties and operations are added to b: (1) a property $state^{sm}$ representing the currently active state of *sm*, and (2) an operation o_t^{sm} for every transition t of *sm*; o_t^{sm} represents the complete behavior of the transition t, including its associated entry and exit actions. Note that this transformation does not consider the behavior of do activities yet. This will be covered later in Section 4.4.

Example 4. *Consider again the given SysML model shown in Fig. 1(a). The corresponding target model including the new description means introduced above is shown in Fig. 1(b).*

As a result of this transformation, each state machine in the original model has been mapped to properties and operations extending its associated block, so that the target model consists of block diagrams and block specifications only. In the paragraphs below, the transformation details for mapping a given state machine to block properties and operations are explained.

[3]Note that state machines have been choosen as a representative to illustrate the first step. However, the corresponding model-to-model transformation can similarly be applied for other notations as well.

4.2 Transformation of State Machines

State machines are transformed into block operations, one for each state machine transition. In general, the overall effect of executing a state machine transition is composed of the effects specified in exit actions, entry actions, and actions directly associated with the transition arrow (the latter is called the *effect of the transition* (Object Management Group, 2011a, 15.3.14), not to be confused with the overall effect described here). If these contributing actions are specified by means of pre-/post-conditions, their overall effect is given by the so-called *relational composition* of these conditions: let $\lhd_{a_1}, \rhd_{a_1}, \lhd_{a_2}, \rhd_{a_2}$ be the pre- and post-conditions associated with actions a_1, a_2, referring to the pre- and post-states of the properties x, y. Then, the condition whether an action $a_1; a_2$ may be invoked is specified by the following pre-condition:

$$\lhd_{a_1;a_2} \equiv \lhd_{a_1} \wedge \exists x', y' : \rhd_{a_1}[x'/x, y'/y] \wedge \lhd_{a_2}[x'/x, y'/y]$$

For any predicate p, the notation $p[x'/x]$ denotes p with every free occurrence of x replaced by symbol x'. Intuitively speaking, $a_1; a_2$ can fire in any pre-state satisfying the pre-condition of a_1, and for which at least one a_1-post-state exists, such that the pre-condition of a_2 holds, and, consequently, a_2 can be executed. The corresponding post-condition for $a_1; a_2$ is specified by

$$\begin{aligned} \rhd_{a_1;a_2} \quad \equiv \quad & \exists x', y' : \\ & \rhd_{a_1}[x'/x, y'/y] \wedge \\ & \lhd_{a_2}[x'/x, y'/y] \wedge \rhd_{a_2}[x'/x@pre, y'/y@pre] \end{aligned}$$

That is, there exists an intermediate state x', y' which is a post-state of a_1 and satisfies the pre-condition of a_2. Then, the post-condition of $a_1; a_2$ coincides with the one of a_2, with the pre-states replaced by the post-states x', y' of a_1.

Example 5. *Suppose* $\lhd_{a_1} \equiv x > 0$, $\rhd_{a_1} \equiv y > 1/x \wedge x = x@pre$, $\lhd_{a_2} \equiv y > 0$, *and* $\rhd_{a_2} \equiv y = y@pre + x \wedge x = x@pre$. *Then*

$$\begin{aligned} \lhd_{a_1;a_2} &\equiv x > 0 \wedge \exists x', y' : y' > 1/x' \wedge x' = x \wedge y' > 0 \\ &\equiv x > 0 \end{aligned}$$

$$\begin{aligned} \rhd_{a_1;a_2} &\equiv \exists x', y' : \\ &\quad y' > 1/x' \wedge x' = x@pre \wedge \\ &\quad y' > 0 \wedge y = y' + x \wedge x = x' \\ &\equiv y > 1/x + x \wedge x = x@pre \end{aligned}$$

Note that for this transformation, we assumed that all actions were specified by means of pre- and post-conditions. However, if this is not the case (e.g. if an action is specified by means of explicit programming statements), *predicate transformers* as described in (Huang et al., 2013, Section 11.1.3) can be used to determine the weakest pre-condition as well as the strongest post-conditions from these statements.

Example 6. *Suppose an action* a_1 *has been specified in the C-programming style as follows:*

```
y = 1/x;
y = y + 1;
```

Then, $\lhd_{a_1} \equiv x \neq 0$ *is the weakest pre-condition and* $\rhd_{a_1} \equiv y = 1/x + 1 \wedge x = x@pre$ *the strongest post-condition of* a_1.

Having represented the actions in terms of pre- and post-conditions, we are now in the position to translate a complete state machine transition. For the start it is assumed that machine *sm* consists solely of simple states – composite states are considered in Section 4.3. A transition t from a state $t.source$ can be performed if (1) the state machine currently resides in the source state $t.source$, and (2) the transition guard $t.guard$ is satisfied. The transition eventually leads to a successor state $t.target$. During this transition, the exit-action of $t.source$, the transition effect $t.effect$, and the entry action of $t.target$ are executed. The resulting pre-condition is

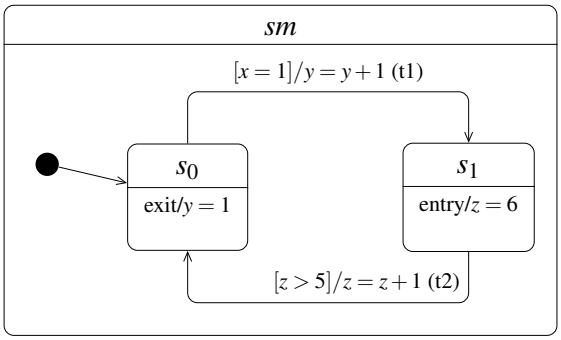
$$\lhd_{o_t^{sm}} \equiv state^{sm} = t^{sm}.source \wedge t^{sm}.guard$$

For the post-condition, we assume that the state machines are well-defined in the sense that the pre-conditions of all entry actions, exit actions, and transition effects involved are fulfilled if $\lhd_{o_t^{sm}}$ holds. Therefore we can specify the post-condition of the transition's *overall* effect as

$$\begin{aligned} \rhd_{o_t^{sm}} \quad \equiv \quad & \rhd_{t^{sm}.source.exit;t^{sm}.effect;t^{sm}.target.entry} \wedge \\ & state^{sm} = t^{sm}.target \end{aligned}$$

That is, executing the transition t^{sm} has the effect of an operation o_t^{sm} whose post-condition is given by the relational composition of the exit action from the source state, the action directly associated with the transition arrow, and the entry action of the target state. As a result, the state machine now resides in the target state.

Example 7. *Fig. 2(a) shows a state machine composed of two simple states* s_1, s_2 *and two transitions. Applying the translation described above leads to the pre- and post-conditions shown in Fig. 2(b).*



(a) State machine *sm*.

$$\lhd_{o_{t_1}^{sm}} \equiv state^{sm} = s_0^{sm} \wedge x = 1$$
$$\rhd_{o_{t_1}^{sm}} \equiv state^{sm} = s_1^{sm} \wedge y = 2 \wedge z = 6$$
$$\lhd_{o_{t_2}^{sm}} \equiv state^{sm} = s_1^{sm} \wedge z > 5$$
$$\rhd_{o_{t_2}^{sm}} \equiv state^{sm} = s_0^{sm} \wedge z = z@pre + 1$$

(b) Resulting pre- and post-conditions.

Figure 2: Transformation of state machines.

4.3 Transformation of Composite States

We now introduce the transformation of single-region composite states (Object Management Group, 2011a, 15.3.11). Their behavioral semantics has already been described informally in Section 2; we will now construct the pre- and post-conditions associated with their outgoing transitions.

Pre-condition. Let t^{sm} be a transition of state machine *sm*. In the following pre-condition formula, let $S = subs(t^{sm}.source)$ denote the proper substates of the transition's source state, and $S_0 = subs(t^{sm}.source) \cup \{t^{sm}.source\}$ the set of substates including $t^{sm}.source$. With these abbreviations, the pre-condition for a transition execution can be specified as follows.

$$\begin{aligned} \lhd_{o_t^{sm}} \quad \equiv \quad & state^{sm} \in S_0 \wedge t^{sm}.guard \wedge \\ & \forall s \in S, u \in s.outgoing : \neg u.guard \end{aligned}$$

Consistent with the informal description of hierarchic state machines in Section 2, this condition states that, in order for t^{sm} to be executed, (1) the innermost simple active state $state^{sm}$ must coincide with, or be a proper substate of $t^{sm}.source$, (2) the guard condition $t^{sm}.guard$ must evaluate to true, and (3) none of the transitions emanating from lower-level states of $t^{sm}.source$ (if any) must be enabled.

Post-condition. For specifying the post-condition, the following recursive functions are used for calculating the innermost simple target state to be reached by a transition, and for collecting the actions to be executed during a transition execution.

Function $\rho_s(t^{sm}.target)$ calculates the target state to be reached if t^{sm} fires.

$$\rho_s(s) \equiv \begin{cases} s & subs(s) = \varnothing \\ \rho_s(sub(s).t.target) & otherwise \end{cases}$$

Starting with the target state $s = t^{sm}.target$ of the transition, it is checked whether the state is simple. If so, the simple innermost target state of the transition execution has been reached. Otherwise the uniquely defined post-state of the submachine's pseudo initial state is used in another recursion of ρ_s.

Let $lca = LCA(t^{sm}.source, t^{sm}.target)$ be the least common ancestor of the transition's source and target states. Function $\rho_\uparrow(s_0, lca, seq = \varepsilon)$ calculates the sequence *seq* of exit actions to be performed when leaving the innermost active simple source state s_0 associated with $t^{sm}.source$ and ending at the state underneath the least common ancestor of *lca*. The recursion starts with the empty sequence $seq = \varepsilon$.

$$\rho_\uparrow(s, lca, seq = \varepsilon) = \begin{cases} seq & \text{if } s = lca \\ \rho_\uparrow(parent(s), lca, (seq; s.exit)) & \\ & \text{otherwise} \end{cases}$$

Let *sseq* denote the sequence of states in the state configuration from *lca*'s substate down to $t^{sm}.target$. Function $\rho_\downarrow(sseq, seq = \varepsilon)$ generates the sequence of entry actions to be performed when traversing *sseq* until $t^{sm}.target$ is reached.

$$\rho_\downarrow(useq, seq) = \begin{cases} seq & \text{if } useq = \varepsilon \\ \rho_\downarrow(ut, (seq; s.entry)) & \text{if } useq = s \circ ut \end{cases}$$

Finally, $\widetilde{\rho}_\downarrow(t^{tm}.target, seq = \varepsilon)$ calculates the entry actions to be performed after having entered $t^{tm}.target$ until the innermost simple target state of t^{tm}'s execution has been reached.

$$\widetilde{\rho}_\downarrow(s, seq) = \begin{cases} seq & \text{if } subs(s) = \varnothing \\ \widetilde{\rho}_\downarrow(sub(s).t.target, (seq; B)) & \\ & \text{otherwise, with} \\ B = sub(s).t.effect; sub(s).t.target.entry & \end{cases}$$

Using these auxiliary functions, the sequence of actions accompanying the overall effect of t^{sm}'s execution is

$$A(s_0) = \rho_\uparrow(s_0, lca, \varepsilon); t^{sm}.effect; \\ \rho_\downarrow(sseq, \varepsilon); \widetilde{\rho}_\downarrow(t^{tm}.target, \varepsilon)$$

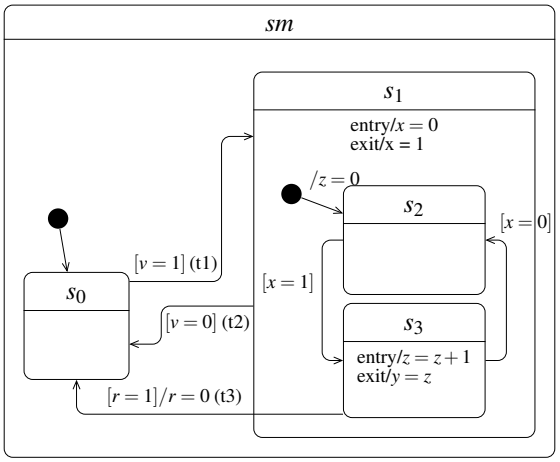
provided that the simple active source state equal to or sub-ordinate to $t^{sm}.source$ equals s_0.

With these preliminaries the post-condition of the overall effect is specified by

$$\triangleright_{o_t^{sm}} \equiv state^{sm} = \rho_s(t^{sm}.target) \wedge \bigvee_{s_0 \in S_0} \left(s_0 = state^{sm}@pre \wedge \triangleright_{A(s_0)}\right)$$

The new active state configuration has the simple state calculated by $\rho_s(t^{sm}.target)$. If s_0 is the uniquely determined simple active pre-state asociated with $t^{sm}.source$, then the post condition derived from the relational composition of the actions which are part of $A(s_0)$ is the second conjunct of $\triangleright_{o_t^{sm}}$.

Example 8. *Fig. 3(a) shows a state machine including one simple state s_0 and one composite state s_1, which, in turn, is composed of two simple states s_2 and s_3. Applying the translation described above leads to the pre- and post-conditions shown in Fig. 3(b).*



(a) Given state machine *sm*.

$$\begin{array}{lcl} \triangleleft_{o_{t1}^{sm}} & \equiv & state^{sm} = s_0 \wedge v = 1 \\ \triangleright_{o_{t1}^{sm}} & \equiv & state^{sm} = s_2 \wedge x = 0 \wedge z = 0 \\ \triangleleft_{o_{t2}^{sm}} & \equiv & state^{sm} \in \{s_2, s_3\} \wedge v = 0 \wedge \\ & & (state^{sm} = s_2 \Rightarrow x \neq 1) \wedge \\ & & (state^{sm} = s_3 \Rightarrow x \neq 0) \\ \triangleright_{o_{t2}^{sm}} & \equiv & state^{sm} = s_0 \wedge \\ & & \big((state^{sm}@pre = s_2 \wedge x = 1) \vee \\ & & (state^{sm}@pre = s_3 \wedge y = z \wedge x = 1)\big) \\ \triangleleft_{o_{t3}^{sm}} & \equiv & state^{sm} = s_3 \wedge r = 1 \\ \triangleright_{o_{t3}^{sm}} & \equiv & state^{sm} = s_0 \wedge y = z \wedge x = 1 \wedge r = 0 \\ \ldots & \ldots & \ldots \end{array}$$

(b) Resulting pre- and post-conditions.

Figure 3: Transformation of composite states.

4.4 Do Activities

Do activities are activated when entering a new active state configuration $s_0.s_1 \ldots s_k = sm$ with active simple state s_0. They remain active until they terminate or until the active configuration changes: if a transition results in exiting states $s_0 \ldots s_{k-j}, j \geq 0$, while

$s_{k-j+1} \ldots s_k$ are also part of the new active configuration, then the do activities specified for $s_0 \ldots s_{k-j}$ are terminated, while the do activities specified for $s_{k-j+1} \ldots s_k$ remain active.

The transformation rules specified in Section 4.3 implicitly cover do activities activated from simple states and specified by means of submachines: their behavior is identical to that of composite states (Object Management Group, 2011a, p. 560). Note, however, that when several do activities defined in simple states $s_1, \ldots, s_n$ reference the same submachine *sm*, the transformation described above will introduce *n copies* of *sm* and add them as single regions to the states $s_1, \ldots, s_n$, which automatically become composite states.

4.5 Model Inputs

Inputs to the model are represented by operations writing to one or more input properties only, with pre-conditions 'true' and with post-conditions that only require that the new value is in the legal range of the variable type.

Example 9. *Consider again the state machine from Figure 3(a) and assume that v is an input to the model, with type integer. The associated input operation is specified by*

$$\begin{aligned} \lhd_v &\equiv \mathit{true} \\ \rhd_v &\equiv v \in \mathit{Integer} \end{aligned}$$

5 MODEL-TO-TEXT TRANSFORMATION: THE TRANSITION RELATION

In this section the generation of the transition relation in propositional form Φ_M is explained. It comes as no surprise that more than one possibility for generating such a formula exists: different paradigms of concurrency, time, and the atomicity of actions are available. Each selection of paradigms leads to another instance of Φ_M. The UML has taken this variety of paradigms into account by introducing *semantic variation points* in its language specification (Object Management Group, 2011a); since the interpretation of SysML state machines is identical to that of the "standard" state machines (so-called *behavioral state machines*) defined by the UML, all the semantic variations available there apply to SysML as well.

Let M be the model resulting from the transformation described in Section 4. Then M consists of block diagrams only, and each block contains the operations defined in the original model plus the new operations generated during the model-to-model transformation.

5.1 Interleaving Semantics

Blocks in M created from different state machines during the model-to-model transformation described above execute concurrently. This is the most liberal assumption; more restrictive ones can be enforced in the original model by introducing synchronization properties that are observed by the state machines involved. Here we choose the *interleaving semantics* paradigm, where only one block operation is executed at a time. As a consequence, each operation execution is automatically atomic, but the sequence of operation executions is nondeterministic: from any block, any operation whose pre-condition evaluates to true can fire. Racing conditions cannot occur, because simultaneous writes to the same property never happen, but the resulting value of a property may be nondeterministic, due to different operation execution sequences. We interpret post-conditions in such a way that all variable symbols *not* occurring in the post condition or only occur with decoration '@pre' remain unchanged by the operation. Non-deterministic operations that might change a property in an arbitrary way will at least state that the changed property value is still an element of the property type. By $W(op)$ we denote the set of variables that are written to according to the post-condition $\rhd_{op}$. If we decorate every free variable occurring in a pre-condition $\lhd_{op}$ by an '@pre' suffix, the resulting proposition is denoted by $\lhd_{op}@pre$. By $V(M)$ we denote the set of all property symbols occurring in M. Let $B(M)$ denote the blocks associated with state machines and $Op(b)$ the operations of block b implementing state machine transitions or model inputs. Auxiliary proposition $S(H)$ ("stable") states that all variables from set H remain unchanged during a transition:

$$S(H) = \bigwedge_{v \in H} v = v@pre$$

With these preparations, the transition relation is written as follows.

$$\Phi_M \equiv \bigvee_{b \in B(M)} \bigvee_{op \in Op(b)} \left(\lhd_{op} @pre \wedge \rhd_{op} \wedge S(V(M) - W(op)) \right)$$

The transition relation is a disjunction over all blocks representing state machines and all operations therein that represent transitions. Every operation whose pre-condition evaluates to true corresponds to a transition that is enabled. The interleaving semantics triggers any of these. Observe that at least one of the disjuncts in Φ_M always evaluates to true, since input operations are always enabled.

5.2 Semantic Variations

Due to the usual space limitations, the detailed exposition of concurrency-related semantic variations is not possible in this paper. The variety of model-to-text transformations that have been constructed in our group support synchronous semantics, dense and discrete realtime mechanisms, and urgent or non-urgent transition behavior. Note that the respective constraints need to be generated only once and, afterwards, can simply be re-used.

6 PROOF OF CONCEPT

The two-step approach to specifying behavioral semantics of SysML specifications has been applied in the development and extension of a model-based testing and bounded model checking tool developed by *Verified Systems International*[4] and currently used in industrial applications in the avionic, automotive, and railway domains. In initial tool versions, the transition relation had been generated in one step, and different generators had been created from scratch for every supported description technique. The novel two-step approach reduces the development time for encoding the behavioral semantics of a new formalism in a considerable way: our effort tracking estimates indicate a reduction of development and verification efforts by at least 40%. This is caused by the fact that new formalisms usually only require a novel model-to-model transformation, while the model-to-text transformations dealing with concurrency aspects can be re-used. Moreover, many verification objectives to be analyzed for the complete transformation can be verified on the level of block diagrams, blocks, and operations, without having to debug the "low-level" transition relation.

7 CONCLUSION

In this paper we have described a two-step approach for generating representations of behavioral SysML model semantics that are suitable for applications in model checking and model-based testing. The first step consists in a model-to-model transformation, and it aims at the equivalent representation of the source formalism (e.g., state machines, sequence diagrams, or activity charts) using block diagrams, blocks, and their properties and operations only. The second step consists in a model-to-text transformation, during which the behavioral semantics of the intermediate model obtained in the first step is represented by means of a transition relation.

Semantic variations concerning the interpretation of the concrete description technique are handled as variants of the first transformation, while semantic variations concerning concurrency and realtime are handled in the second transformation. We currently explore the possibility to specify these semantic variations by means of constraint blocks encapsulated in packages, so that the selection of concrete semantics can be realized by means of package import in combination with parametric diagrams enforcing the constraints associated with the selected semantic variation.

ACKNOWLEDGEMENTS

This work was supported by the Graduate School SyDe (funded by the German Excellence Initiative within the University of Bremens institutional strategy), the European Union within the FP7 project COMPASS under grant agreement no. 287829, the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001, and the German Research Foundation (DFG) within the Reinhart Koselleck project under grant no. DR 287/23-1 as well as a research project under grant no. WI 3401/5-1.

[4]See www.verified.de.

REFERENCES

Beckert, B., Hähnle, R., and Schmitt, P. H. (2007). *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003). Bounded model checking. *Advances in Computers*, 58:117–148.

Brucker, A. D. and Wolff, B. (2006). The HOL-OCL book. Technical Report 525, ETH Zurich.

Cabot, J., Clarisó, R., and Riera, D. (2008). Verification of UML/OCL class diagrams using constraint programming. In *Int'l Conference on Software Testing Verification and Validation Workshop*, pages 73–80, Washington, DC, USA. IEEE Computer Society.

Gogolla, M., Kuhlmann, M., and Hamann, L. (2009). Consistency, independence and consequences in UML and OCL models. In *Tests and Proofs*, pages 90–104.

Hilken, C., Seiter, J., Wille, R., Kühne, U., and Drechsler, R. (2014). Verifying Consistency between Activity Diagrams and Their Corresponding OCL Contracts. In *Forum on Specification & Design Languages*.

Huang, W., Peleska, J., and Schulze, U. (2013). Test automation support. Technical Report D34.1, COMPASS Comprehensive Modelling for Advanced Systems of Systems. Available under *http://www.compass-research.eu/deliverables.html*.

Object Management Group (2010). OMG Systems Modeling Language (OMG SysML^{TM}). Technical report, Object Management Group. OMG Document Number: formal/2010-06-02.

Object Management Group (2011a). OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. Technical report, OMG.

Object Management Group (2011b). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. Technical report, Object Management Group. OMG Document Number: formal/2011-06-02.

Object Management Group (2012). OMG Object Constraint Language (OCL). Technical report, Object Management Group. OMG Document Number: formal/2012-01-01.

Peleska, J. (2013). Industrial-strength model-based testing - state of the art and current challenges. In Petrenko, A. K. and Schlingloff, H., editors, Proceedings Eighth Workshop on Model-Based Testing, Rome, Italy, 17th March 2013, volume 111 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–28. Open Publishing Association.

Soeken, M., Wille, R., and Drechsler, R. (2011). Verifying dynamic aspects of UML models. In *Design, Automation and Test in Europe*, pages 1077–1082. IEEE Computer Society.