Mapping UML to Labeled Transition Systems for Test-Case Generation

A Translation via Object-Oriented Action Systems[⋆]

Willibald Krenn[1], Rupert Schlick[2], and Bernhard K. Aichernig[1]

[1] Institute for Software Technology, Graz University of Technology, Austria
{wkrenn,aichernig}@ist.tugraz.at
[2] Austrian Institute of Technology, Vienna, Austria
Rupert.Schlick@ait.ac.at

Abstract. The Unified Modeling Language (UML) is a well known and widely used standard for building software models. While it is familiar to many software engineers, it lacks standardized formal semantics. In this paper, we extend on the formalism of object-oriented action systems (OOAS) and describe a mapping of a selected UML-subset to OOAS by choosing one of the several possible semantics of UML. This mapping, together with the introduction of a trace semantics for OOAS, paves the way for applying tools for and theory of labeled transition systems to UML-models. As a running example, we use a car alarm system in the context of model-based test-case generation and show how the UML mapping is done.

1 Introduction

Today, embedded computer systems constitute an integral part of almost all technology surrounding us. They are increasingly integrated in safety-relevant systems, either in any kind of vehicles, medical equipment, or industrial or public control systems. Evidently, any possible measure has to be taken to ensure the dependability of such systems, from early planning and design to final installation and maintenance.

The standards EN 50128 and IEC 61508 recommend the use of formal methods, especially at higher Safety Integrity Levels (SILs). However, despite the decades of research dedicated to formal methods, most engineers still lack experience and confidence in this field. Techniques like theorem proving or model checking are rarely applied to large and complex systems.

Therefore, testing remains the preferred method of verification, despite the fact that it is very expensive. In general, about half of the overall effort of a project is dedicated to testing, and for safety-relevant projects the amount of time spent on testing is even higher. Consequently, there is a huge demand for

[⋆] Research herein was funded by the EU FP7 project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES).

reliable automatic test case generation tools grounded on solid foundations. The European FP7 project MOGENTES serves these demands.

MOGENTES stands for Model-based Generation of Tests for Dependable Embedded Systems and its goal is to significantly enhance testing and verification of dependable embedded systems by means of automated generation of efficient test cases relying on development of new approaches as well as innovative integration of state-of-the-art techniques. In particular, MOGENTES aims at the application of these technologies in large industrial systems, simultaneously enabling application domain experts (with rather little knowledge and experience in usage of formal methods) to use them with minimal learning effort.

The industrial partners in the project identified UML as their future modeling paradigm and hence, require test case generation tools to process UML models. This need conflicts with the requirement that our test case generation technique has to be build on solid foundations, because UML lacks a standard formal semantics. However, a formal semantics is essential for our testing techniques based on precise fault-models and formal notions of conformance. Therefore, we decided to treat UML as a front-end modeling language and translate it to a formal back-end formalism on which our test case generators will work on.

In this paper, we give insights into this translation process. A car alarm system serves as a running example. Section 2 presents the UML model of the car alarm system including the technique to express the testing interface in UML class diagrams. Then, Section 3 presents and motivates our back-end formalism, namely Object-Oriented Action Systems (OOAS), a formalism well-suited for expressing object-oriented models of embedded systems. This section also presents a further level of semantic mapping: the behavior of state-rich OOAS is interpreted as a series of controllable and observable events. It is this event-level on which our test case generators work. This gives us the advantage that we can base our formal testing approach on the existing testing theory on labeled-transition systems. Next, in Section 4 we discuss our semantic mapping, including the translation of non-trivial UML state charts with nested states, parallel regions and time triggers. In Section 5 we discuss the case study. Finally, in Section 6 we draw our conclusion and give an outlook on future and related work.

2 A UML-Model

We use a very simplified car alarm system as an example for discussing the concepts and issues of the transformation of UML models to action systems. The example is taken from Ford's automotive demonstrator within MOGENTES and the main purpose of this rather simple example within the project is to test and validate the test-case generation work flow on a basic level. Notice that we are dealing with black-box testing here, as, e.g., Ford wants to test components provided by an external partner based on the requirements that were given to this company.

Before we can generate any test-cases, we need to build a model from the requirements. For our simplified car alarm system (CAS), we were given the following three textual requirements.

Requirement 1: Arming. The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment and all doors are closed.

Requirement 2: Alarm. The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment or the bonnet. The hazard flasher lights will flash for five minutes.

Requirement 3: Deactivation. The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

When trying to construct an animated model based on textual requirements it is often the case that conflicts or underspecified situations become apparent. One might think that the simplistic car alarm system is sufficiently described by these three textual requirements – the contrary is the case. What is left unspecified is the case of what happens when an alarm is ended by the five minute timeout: does the system go back to armed directly, or does it need to wait for all doors to be closed again before returning to armed?

2.1 Testing Interface and Instantiation

The UML model of the car alarm system comprises four classes and four signals, as shown in Fig. 1. The class *AlarmSystem* is marked as system under test (SUT) and may receive any of the *Lock*, *Unlock*, *Close*, or *Open* signals. At the same time, the SUT calls methods of the classes *AlarmArmed*, *AcousticAlarm*, and *OpticalAlarm* – all of them marked as being part of the environment.

Notice that the context diagram specifies the observations (all calls to methods being part of the environment) we can make and the stimuli the system under test can take (all signals). In effect, this diagram specifies our testing interface.

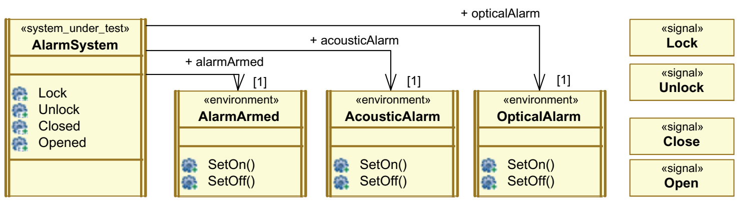

Fig. 1. Car Alarm System - Testing Interface

We use an initialization diagram (not shown) to specify the system configuration: we create a singleton object for each of the classes.

2.2 State Machine

Fig. 2 shows the CAS state-machine diagram. From the state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car.

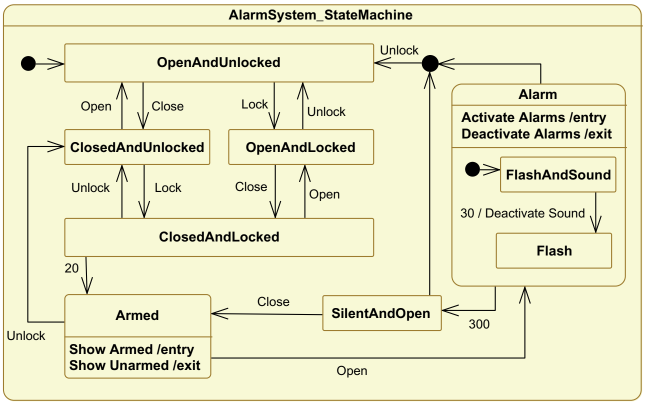

Fig. 2. Car Alarm System - State Machine

Actions of closing, opening, locking, and unlocking are modeled by corresponding signals *Close*, *Open*, *Lock*, and *Unlock*. As specified in the first requirement, the alarm system is armed after 20 seconds in *ClosedAndLocked*. Upon entry of the *Armed* state, the model calls the method *AlarmArmed.SetOn*. Upon leaving the state, which can be done by either unlocking the car or opening a door, *AlarmArmed.SetOff* is called. Similar, when entering the *Alarm* state, the optical and acoustic alarms are enabled. When leaving the alarm state, either via a timeout or via unlocking the car, both acoustic and optical alarm are turned off. When leaving the alarm state after a timeout (cf. second requirement) we decided to interpret the requirements in a way that the system returns to an armed state only in case it receives a close signal. Turning off the acoustic alarm after 30 seconds, as specified in the second requirement, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

2.3 Semantic Variation Points

Despite being machine readable, the presented UML model lacks precise semantics. As an example, the event processing-machinery within a state machine is not fully specified within the UML standard:

> No assumptions are made about the time intervals between event occurrence, event dispatching, and consumption. This leaves open the possibility of different semantic variations such as zero-time semantics. It is a semantic variation whether an event is discarded if there is no appropriate trigger defined for them. *[1], Trigger, p. 456*

These intentionally underspecified areas in the UML standard are called “semantic variation points” and are used within the UML specification to *provide leeway for domain-specific refinements of the general UML semantics ([1], p. 17).* Other semantic variation points affect, e.g., time events and signal events. When we discuss the mapping of UML to object-oriented action systems in Section 4, we will make use of these semantic variation points and define one particular behavior of our models. Note that these choices may form the basis for semantic tests: when creating test cases from UML models we may want to find implementations that violate our chosen interpretation of a semantic variation point! Note also that in addition to semantic variation points, there are other sources of non-determinism in the UML specification, e.g., transition firing (cf. *[1], State Machine, p. 566*).

3 Object-Oriented Action Systems

In this section we present object-oriented action systems, our intermediate-level modeling language that we use for giving UML models precise semantics. Object-oriented action systems are an extension to the action system formalism initially proposed by Back et al. in [2,3]. The object-oriented extension presented here is based on the work of Bonsangue et al., published in [4]. We also use a prioritized composition operator that has already been introduced by Sekerinski et al. in [5]. Notice, however, that our work is the first to combine object-oriented action systems (with custom extensions), prioritized composition, complex data types, and a trace semantics of action systems. We start our description of object-oriented action systems with the introduction of normal (non-object-oriented) action systems.

3.1 Action Systems

Syntactically, we may represent an action system AS comprising m functions, a named actions, d non-deterministically composed anonymous actions, and a set F_I of imported functions syntactically as follows.

$$\begin{array}{lll} AS =_{df} |[& \textbf{var} & V : T = I \\ & \textbf{functions} & F_n^1 = F_b^1; \ldots; F_n^m = F_b^m \\ & \textbf{actions} & N_n^1 = N_b^1; \ldots; N_n^a = N_b^a \\ & & \textbf{do}\ A_1 \,\Box\, ... \,\Box\, A_d\ \textbf{od} \\ ]| : F_I & & \end{array}$$

Notice that functions have a name, a body and may return a value. Named actions are similar to functions but may not have a return value. In the remainder of this paper, we assume that named actions may only be called from within the **do od**-block (that is, not from within named actions or functions), and that function-calls may not be recursively nested. We also demand that each named action has the form of a guarded command. Relying on these assumptions, we are allowed to re-write the action system in a more classical form, where only the actions within the **do od**-block are left:

$$\begin{array}{ll} AS =_{df} |[& \textbf{var}\ V : T = I \\ & \quad \textbf{do}\ A_1 \,\Box\, ... \,\Box\, A_d\ \textbf{od} \\ ]| : Z & \end{array}$$

Within this representation, V is a vector of variables of types T, initialized with values I, all A_i ($1 \leq i \leq d$) are actions, and $\Box$ stands for non-deterministic, demonic choice. Demonic choice of actions means that when an aborting action is enabled, this action is chosen. Notice that after "inline'ning" all imported functions $\in F_I$, Z denotes the set of imported variables of the environment that was accessed by the imported functions.

After eliminating all function calls, the action system consists of basic actions only (cf. Table 1) and all actions are part of the **do od**-block, also known as Dijkstra's guarded iteration statement [6]. The guarded iteration statement can be thought of as being a loop that selects one enabled action A_i for execution in each iteration. In case there is no action enabled, execution of the action system ceases as execution of the loop terminates.

We can determine the enabledness of an action A_i by computing the enabledness guard: Because we do not want an action A_i to be enabled for states in which it is guaranteed to establish any postcondition, i.e. behave miraculously, the enabledness guard is defined as $g.A =_{df} \neg wp(A, false)$, where $wp : Action \times (State \mapsto Bool) \mapsto (State \mapsto Bool)$ is the weakest precondition predicate transformer. For example, the precondition of a guarded command is given by

$$wp(\text{requires } guard : A \text{ end}, q) \equiv guard \Rightarrow wp(A, q)$$

with "$\Rightarrow$" denoting logical implication. Table 1 lists the weakest preconditions of all actions for any given predicate q.

Table 1. Semantics of Basic Actions

Action	Notation	$wp(Action, q)$
Sequential Composition	$S_1; \ldots; S_n$	$wp(S_1, wp(\ldots, wp(S_n, q)))$
Nondeterministic Composition	$S_1 \Box S_2$	$wp(S_1, q) \wedge wp(S_2, q)$
Prioritizing Composition	$S_1 // S_2$	$wp(S_1, q) \wedge$ $(\neg g.S_1 \Rightarrow wp(S_2, q))$
Guarded Command	requires p: S_1 end	$p \Rightarrow wp(S_1, q)$
Multiple Assignment	$y := e$	$q[y := e]$
Nondeterministic Assignment	$z := z'$ with Q	$(\forall z' \in Q.z \cdot q[z := z'])$
Local Variables	var $x_1 : T_1; \ldots; x_n : T_n : S$	$\forall x_1 \ldots x_n : wp(S, q)$
Skip	**skip**	q
Abort	**abort**	$false$

In Table 1, S_1, S_2 each denote an action, y lists of variables, z, z' variables, e is a list of expressions, p is a predicate over the state of the action system, $g.S_1$ is the enabledness guard of action S_1, and Q is a predicate over z, z' (and the state). The nondeterministic assignment assigns to variable z the value of z' for which Q holds. The statement aborts if this is not possible [3]. Notice that an action will terminate if the *termination guard* $t.A = wp(A, true)$ holds.

For any of the defined actions, the monotonicity (1) and conjunctivity (2) properties hold:

$$(p \Rightarrow q) \implies (wp(A,p) \Rightarrow wp(A,q)) \tag{1}$$

$$wp(A,P) \wedge wp(A,Q) \equiv wp(A,P \wedge Q) \tag{2}$$

In addition, we require an action to be bounded non-deterministic. The parallel composition of two action systems is done by joining all actions and variables. (Some variables may be shared between the systems.) As an example, the parallel composition $AS^1 \parallel AS^2$ of two action systems

$$\begin{aligned} AS^1 = |[\ & \mathbf{var}\ X : T^1 = I^1; \\ & \mathbf{do}\ A^1_1 \Box \ \dots \ \Box A^1_m\ \mathbf{od}]| : u^1 \\ AS^2 = |[\ & \mathbf{var}\ Z : T^2 = I^2; \\ & \mathbf{do}\ A^2_1 \Box \ \dots \ \Box A^2_n\ \mathbf{od}]| : u^2 \end{aligned}$$

yields $AS^{1\|2}$:

$$\begin{aligned} AS^{1\|2} = |[\ & \mathbf{var}\ X : T^1 = I^1; Z : T^2 = I^2; \\ & \mathbf{do}\ A^1_1 \Box \ \dots \ \Box A^1_m \\ & \Box A^2_1 \Box \ \dots \ \Box A^2_n\ \mathbf{od} \\ &]| : (u^1 \cup u^2) \setminus (v^1 \cup v^2) \end{aligned}$$

where v^i denotes all variables used (exported) from action system AS^i.

3.2 Object Orientation

We use the work of Bonsangue et al. [4] as the basis for object-oriented action systems: in particular we share the transformation step from object-oriented action systems to action systems. We differ in the notion of named actions and procedures and we add the ability to prioritize objects of a particular class with respect to objects of another class. Within our methodology, we use a very simple form of inheritance: A class C^2 is a valid subclass of C^1 if and only if the (syntactic) superposition (cf. [7]) refinement holds between the classes. Roughly speaking this means that C^2 may introduce additional variables and actions. However, none of the additional actions may have any effect on the variables of C^1, it must be guaranteed that when only considering the new actions and the initial state the system terminates, and the exit condition of C^2 must imply the exit condition of C^1. The subclass C^2 may override (refine) actions of C^1 in a way that the guard is strengthened and values to the additional variables are assigned.

Like most object-oriented programming languages, objects are constructed at runtime from classes with the help of a constructor statement $o := new(C)$, where o represents the instance (object) and C stands for some class. Similar to [4], a class C is a named type and can be represented as tuple $C =_{df} (C_n, C_b)$

where $C_n \in \mathcal{CN}$ is a class name from the set of class-names $\mathcal{CN}$ and C_b is the body of the type definition:

$$\begin{array}{lll} C_b =_{df} |[& \textbf{var} & V : T = I \\ & \textbf{methods} & M_n^1 = M_b^1; \ldots; M_n^m = M_b^m \\ & \textbf{actions} & N_n^1 = N_b^1; \ldots; N_n^a = N_b^a \\ & \textbf{do}\ A\ \textbf{od} & \\ \quad]| : M_I & & \end{array}$$

Similar to our definition of action systems, V denotes a vector of state-variables of types T, initialized with a value of I. A class may have m methods, each one having a name and a body: $M^i =_{df} (M_n^i, M_b^i)$ $(1 \leq i \leq m)$. As in action systems, the class may import a set of methods M_I from other classes. Like before, we do not allow for recursive calls of methods (so we can easily in-line the calls), and named actions may only be called from within the **do od**-block. Notice that this implies that methods are "public", as they can be called by any other method or action. Again, methods are free to return a value while named actions may only take input parameters.

We restrict an object-oriented action system to a finite set of classes $\mathcal{C} =_{df} \{C^1, \ldots, C^k\}$ and a finite set of objects. Practically, this means that we allow object-instantiation only during state-variable initialization, which permits us a rather easy check of finiteness. When a class in an object-oriented action system is marked as *autocons*, one instance of the class will be created automatically at system start and is called a "root object".

We assume that all objects of one class have the same priority. Between objects of different classes, however, we allow ordering with the help of the prioritized composition operator: we introduce a so-called system assembling block (SAB). The SAB, which is an extension to the work of [4], specifies the ordering of priorities between objects of different classes. We rely extensively on this feature in order to model, e.g., event broadcasting, as is discussed in Section 4.4. The syntax of the system assembling block is defined by the following grammar.

$$SAB ::= C_n \ ((\ \Box \mid //\ )\ SAB)?$$

Notice that the non-deterministic choice operator denotes parallel composition and the prioritizing composition operator expresses a prioritizing composition of objects. As an example, $C^1 // C^2$ means that only if there is no action enabled in any of the C^1 objects, actions of any of the C^2 objects will be looked at.

Hence, we define an object-oriented action system as a 3-tuple $(\mathcal{C}, \mathcal{R}, SAB)$, where $\mathcal{C}$ is a finite set of classes $\{C^1, \ldots C^k\}$, $\mathcal{R} \subseteq \mathcal{C}$ is a set of classes that need to be instantiated once at system start, and SAB is the system assembling block. Within the system assembling block, each class-name $C_n^i \in \mathcal{C}$ must be listed once, and all listed names must be from $\mathcal{CN}$.

The semantics of object-oriented action systems are given by a mapping to action systems which is based on the work presented in [4]. The main idea of the mapping is to create one action system per object and join all action systems as specified in the system assembling block.

After generating the set of all object names $\mathcal{ON} =_{df} \cup_{C^i \in \mathcal{C}} \mathcal{ON}_{C^i}$, in a first step every action of a class C^i is translated into an action of an action system. During this step, method calls are transformed into function calls of action systems. Because a function call in an action system needs to statically specify the target action system name and the function name, i.e. looks like *ActionSystemName.FunctionName*(...), and the name of the target object (action system) is not known until runtime, the translation needs to split the method call into a non-deterministic choice over calls to all possible action systems (objects) created for the target type. Notice that an implementation may do this more efficiently: here we only show how an object-oriented action system could be directly specified using the action system syntax. Also notice that during the transformation of an object-oriented action system all named actions and methods get renamed so that the names are unique.

In a second step, each single class C^i of an object-oriented action system is translated into an action system: for the class C^i itself and for each object of C^i an action system is constructed. Remember that the methods have already been translated in the previous step. All action systems that were built in this step are then parallel composed and form the action system $A(C^i)$ describing class C^i.

Finally composing all action systems $A(C^i)$ as specified in the system assembly block completes the mapping of the object-oriented action system OO to an action system $A(OO)$.

3.3 Prioritizing Composition

Given two actions S_1 and S_2, then the prioritizing composition $S_1 \mathbin{//} S_2$ can be re-written using non-deterministic choice and the enabledness guard as follows (cf. Table 1).

$$S_1 \mathbin{//} S_2 \equiv S_1 \,\Box\, (\text{requires } \neg g.S_1 : S_2 \text{ end})$$

Hence, in case the enabledness guard of action S_1 does not hold, the system will deterministically choose action S_2 provided S_2 is enabled. However, if S_1 is enabled, the system will only choose S_1 because action S_2 is guarded by $\neg g.S_1$.

When prioritizing composition is applied to action systems AS^1 and AS^2 (as in the SAB), it is defined such that priority is given to the actions of AS^1 over the actions of AS^2. As an example, the prioritized composition $AS^1 \mathbin{//} AS^2$ of two action systems

$$\begin{aligned} AS^1 = |[\ &\textbf{var}\ X : T^1 = I^1_0;\\ &\textbf{do}\ A^1_1 \,\Box\, \ldots \,\Box\, A^1_m\ \textbf{od}]| : u^1\\ AS^2 = |[\ &\textbf{var}\ Z : T^2 = I^2_0;\\ &\textbf{do}\ A^2_1 \,\Box\, \ldots \,\Box\, A^2_n\ \textbf{od}]| : u^2 \end{aligned}$$

yields $AS^{1\,//\,2}$:

$$\begin{aligned} A^{1\,//\,2} = |[\ &\textbf{var}\ X : T^1 = I^1_0; Z : T^2 = I^2_0;\\ &\textbf{do}\ (A^1_1 \,\Box\, \ldots \,\Box\, A^1_m) \end{aligned}$$

$$// \, (A_1^2 \,\Box\, \ldots \,\Box\, A_n^2) \ \textbf{od}$$
$$]| : (u^1 \cup u^2) \setminus (v^1 \cup v^2)$$

where v^i denotes all variables exported from action system AS^i.

Like on actions, prioritizing composition is associative on action systems. However, it does not in general distribute over parallel composition to the right when used on action systems. This is due to local variables that would be duplicated.

3.4 Complex Data Types

Finally we add complex data types, such as maps, lists, and tuples (besides objects) to our language of OOAS. Most operators on these complex types were taken from the set of operators defined in the Vienna Development Method (VDM) [8,9] and include domain/range restrictions, and distributed union/intersection among other standard operators. We also allow array-like access of list elements and set operators to be working on lists.

3.5 Trace Semantics

For black-box test-case generation purposes, we are interested in the abstract computation sequences, i.e. traces, of an action system. In [2] the computation of an action system starting from an initial state γ_0 is defined as a possibly infinite sequence t of the form

$$t =_{df} \gamma_0 \xrightarrow{S_i} \gamma_1 \cdots$$

with each $g.S_i$ enabled in the transition's initial state.

We will use the concept of named actions to define more abstract computation traces: we extend the name of named actions to include markers for *observable* and *controllable* actions. All methods and all unmarked actions are considered *internal*. Hence, any name N_n^i of a named action is built according to the following grammar.

$$N_n^i ::= (\ \textit{'obs'} \mid \textit{'ctr'} \mid \textit{' '})\textit{' '} \ \textit{Identifier}$$

Informally, an abstract computation sequence starting from an initial state γ_0 is a possibly empty or infinite sequence t_{abs} of the form

$$t_{abs} =_{df} \gamma_0 \xrightarrow{N_n^i} \gamma_1 \cdots$$

where $\xrightarrow{N_n^i}$ means the application (call) of the action body N_b^i of action N^i when $g.N_b^i$ holds at the transition's initial state or there is some sequence of basic actions (including method calls) $\gamma_j \xrightarrow{S_i} \cdots$ starting at the current state γ_j and leading to a state where $g.N_b^i$ holds.

Notice that the concept of labeled actions can already be found in [2] and that in [10] a similar event-based view of action systems is taken.

4 Chosen UML Semantics

Since many UML constructs represent rather complex behavior, mapping to OOAS means also implementing these constructs in OOAS. Because of the size of UML, not only would full feature support be a major effort, but many elements simply are not useful in the context of behavioral test models. Therefore, we limit the transformation tool to a subset of UML.

In the following subsections, we describe the selected subset along with some motivation and then four of the more interesting aspects of the transformation are discussed, along with the taken decisions regarding semantic interpretation.

4.1 Used UML Subset

In the context of embedded and safety critical systems, modeling with state machines is quite common and fits the needs of the domain. The UML subset supported by the transformation therefore comprises class diagrams, state machines and a subset of OCL. The selection is mainly based on the needs of the demonstrator applications within MOGENTES; some state-machine concepts that were intentionally left out, like deferred triggers and history states, could be added when needed. Table 2 summarizes all supported UML elements. In the table, *"Simple" Inheritance* means that we do not support any polymorphism or late-binding. It is set in brackets as our tool-support for inheritance is not yet complete. Also, while the transformation in principle supports the float data type, we do not use it currently. Method and effect opaque behavior bodies are filled using a minimal custom language that can be used to express signal sending/broadcasting, method calling and assignment. If needed, this small language can easily be replaced by another one, e.g., the Object Action Language (OAL).

In the transformation, we strive for following UML v2.2 Standard. Nonetheless we have made some design choices, aside from the selected elements: object

Table 2. Supported UML Elements

"Types"	Class	Classes	Active/Passive	OCL	and, or, not, implies
	Enums		Associations		=, <, >, <=, >=
	Signal		("Simple" Inheritance)		union, intersect
	Bool		Member Fields		select, collect
	Int		Methods Def. + Body		exists, forall, oclIsInState
	(Float)		Signal Reception		Literals (Numbers, Bool)
State Machines	Substate Machine				
	Orthogonal Regions				
	(Final-, Initial-, Pseudo-) State				
	Entry, Exit Action				
	Transitions with Effects				
	Trigger with Change/Signal/Call/Time Events				
	Constraints (OCL)				
	Junctions, Choice				

instantiation is limited to the initialization phase while destruction of objects is not used at all. This fits well into the current practice in embedded systems design, where a constant, limited and predictable memory footprint is wanted. This also avoids some of the semantic variation points on deletion/creation in context of composition and aggregation relations between classes.

Classes, member-fields and method definitions map easily to the respective counterparts in the OOAS as described in the previous section. Mapping of inheritance is also straight forward, provided the subclass is a valid superposition-refinement of the superclass. Behavioral aspects are mainly expressed with state machines in the selected UML subset; while there is a similarity between state machine transitions and guarded actions, some of the features of UML state machines need some more thought on how to implement them in OOAS.

4.2 Events

Transitions in UML state machines are triggered by events. There are four trigger types: *signal triggers, call triggers, change triggers* and *time triggers*. All events concerning an object are stored in the object's event pool until they are consumed, e.g., by transitions. Although we assume that there is always only one external input event at a time, multiple objects might be interested in an event, e.g., a signal reception event, and processing the event might produce further events before the initial events are consumed in the other objects. Hence the need to implement an event-management logic.

Most models are developed with an assumption of in order processing of events, therefore we decided to use event queues, implemented as lists in OOAS. The event distribution logic of the respective event type adds the event at the end of the list. State machine transitions consume events from the front of the list. Providing real pool behavior with OOAS can be easily done by non-deterministically choosing the event to process next, at the cost of increased non-determinism.

A transition path (one direct transition between states or a series of transitions connected by choice pseudo-states) is implemented as a named action of the following form:

```
transition_OpenAndLocked_to_ClosedAndLocked =
  requires (state = OpenAndLocked) and
           (events <> [nil]) and
           (hd events)[0] = received_AlarmSystem_Close):
    state := ClosedAndLocked
  end;
/* .. other transitions .. */
dequeue =
  requires events <> [nil] :
    events := tl self.events
  end
```

The *requires* expression (guard) of the transition tests if the object is in the source state, and whether the first event in the event queue is one of the trigger events of the transition. In case the transition has a guard, it is also checked. All actions modeling transitions are combined by non-deterministic choice as follows.

```
do
  ( (transition_Armed_to_Alarm;
      call_AlarmArmed_SetOff;
      call_OpticalAlarm_SetOn;
      call_AcousticAlarm_SetOn )
    [] transition_OpenAndLocked_to_ClosedAndLocked
    [] /* .. other transitions .. */
  ) //  dequeue()
od
```

As can be seen, entry and exit actions, e.g., *call_AlarmArmed_SetOff*, are sequentially composed with the transition action. Transition effects, if present, are treated in the same manner. Dequeuing of the event is done in the *dequeue* action by removing the head-element of the event-list. The dequeue action is enabled only if there is no enabled transition left. This allows modeling of events triggering multiple transitions as well as events that enable no transition, as required by the standard (cf. *[1], State Machine, p.566*). We discuss the handling of multiple transitions for one event in detail in Subsection 4.3.

Calls and Signals. After an object has received a signal or a method of the object was called, the corresponding events are added to the object's event queue. (Currently, there is no support for handling synchronous method calls.) We represent these events in the OOAS as data-tuples, hence the event queue is a list of tuples, and is initially empty.

```
types
   t_eventname_AlarmSystem = {__received_AlarmSystem_Close,
        __received_AlarmSystem_Lock, ... }; /*enumeration*/
   t_event_AlarmSystem = (t_eventname_AlarmSystem) /*tuple*/
var
   /* object event queue */
   events : list[7] of t_event_AlarmSystem = [nil]

methods
         /* add a lock event to queue */
   __rcv_Lock =
       events := events ^ [t_event_AlarmSystem(
            __received_AlarmSystem_Lock)]
   end;
```

If there are call parameters and signal properties, the event-type has to be extended to provide place for the event name itself and all parameters. In the example above, there are no properties or parameters, hence *t_event_Alarmsystem* is a 1-tuple.

According to the UML standard, signal transmission might be lossy, out of order, or even allow duplication of signals. As a practical example we may consider a distributed embedded system using the CAN bus: there, message transmission is based on priorities. If we want to model this kind of behavior, we need to explicitly represent it in the UML-model as our transformation guarantees in-order message processing.

4.3 Object Concurrency and Regions

There are two different sources of concurrency in UML models. One source are active classes, the other one are orthogonal regions in state machines. Since the

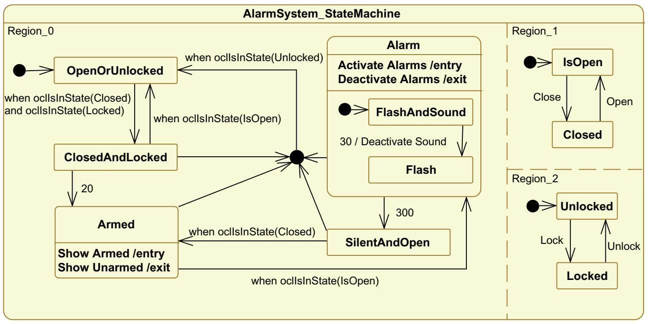

Fig. 3. Car Alarm System - State machine implemented using orthogonal regions

UML standard does not request true parallel execution we decided to use the interleaving semantics provided by the non-deterministic choice operator. Therefore, concurrent execution of active classes can be trivially mapped to a parallel composition of classes of an OOAS (cf. Section 3). The resulting interleavings of the active objects represent all possible "sequentializations".

The second source of concurrency are orthogonal regions of state machines. State machines and states may be split into two or more parallel active regions, so called orthogonal regions. To discuss our support of orthogonal regions, we extend the testing-model presented in Section 2. Instead of permitting an input signal, e.g., *Open*, only at certain places, we take a more realistic view and add two orthogonal regions. Each of the new regions has two states and the state machine may flip between these states when encountering a matching input signal. This may happen at any time and "runs" in parallel with the main-logic of the car-alarm-system. Fig. 3 shows the resulting state machine that includes all behavior that was possible in our first CAS-version. Notice that the UML standard does not specify the order in which parallel enabled and selected transitions have to fire (cf. *[1], State Machine, p. 566*). Hence we are allowed to map this type of concurrency to non-deterministic choice over enabled transitions again.

Since we need to memorize the state of each region of the state machine (notice that sub-states of a state automatically lie in a separate region) we need to introduce a state variable for every region in the OOAS. Below we sketch the state variable definitions for the CAS with regions.

```
types
    /* enumeration types for class state variables */
    AlarmSystem_Region_2 = {AlarmSystem_Region_2_Unlocked , ...};
    AlarmSystem_Region_0__Alarm__Region_0 = {
        AlarmSystem_Region_0__Alarm__Region_0_Initial_0 , ...};
    AlarmSystem_Region_0 = {AlarmSystem_Region_0_Initial_0 , ...};
    AlarmSystem_Region_1 = {AlarmSystem_Region_1_Initial_0 ,
        AlarmSystem_Region_1_IsOpen , AlarmSystem_Region_1_Closed}
var
    /* class state variables */
    Region_2 : AlarmSystem_Region_2 = AlarmSystem_Region_2_Initial_0;
    Region_0__Alarm__Region_0 : AlarmSystem_Region_0__Alarm__Region_0
        = ...;
    Region_0 : AlarmSystem_Region_0 = ...;
    Region_1 : AlarmSystem_Region_1 = ...;
```

The state of the object is made up from all region states with an active parent region. Sub regions of inactive states are ignored for the moment, but can be used to support history pseudo states. The transitions from *Locked* to *Unlocked* and from *FlashAndSound* to *Flash* in the CAS are translated to the following:

```
transition_Locked_to_Unlocked_Transition_0 =
  requires ((Region_2 = AlarmSystem_Region_2_Locked)
     and (not __consumed_Region_2) and (events <> [nil])
     and ((hd events)[0] = __received_AlarmSystem_Unlock)) :
    Region_2 := AlarmSystem_Region_2_Unlocked;
    /* set consumed flags */
    __consumed_Region_2 := true
  end;
transition_FlashAndSound_to_Flash_Transition_0 =
  requires ((Region_0__Alarm__Region_0 =
       AlarmSystem_Region_0__Alarm__Region_0_FlashAndSound)
       and (Region_0 = AlarmSystem_Region_0_Alarm)
       and (not __consumed_Region_0__Alarm__Region_0)
       and (events <> [nil])
       and ((hd events)[0] = __time_trigger_FlashAndSound_30_Flash)) :
    Region_0__Alarm__Region_0 :=
        AlarmSystem_Region_0__Alarm__Region_0_Flash;
    /* set consumed flags */
    __consumed_Region_0__Alarm__Region_0 := true
  end
```

The consumed flags in the OOAS-code are necessary to guarantee standard-conforming behavior of transitions in orthogonal regions that are triggered by the same event. When the *consumed* flag for a region is set, it means that the event has already been processed by the region. Hence the transitions of the region depending on events must not fire as long as the flag is set: each transition tests for its region's consumed flag being false and sets it when it is done. The *dequeue* action then resets all flags. Transitions in sub-states may also consume events for the region and in fact have priority over transitions with the same trigger in higher level regions. To ensure this behavior, before processing the next transition, the consumed flags for regions whose child regions have the flag set, are set. Furthermore, transitions within sub-state machines are put first in a prioritized composition to mirror the priority of transitions given by the standard, as can be seen below.

```
actions
  mark_Region_0_conditional_consumed =
    requires __consumed_Region_0 = false
        and (__consumed_Region_0__Alarm__Region_0):
      __consumed_Region_0 := true
    end;
  /* other actions */
do
    ( transition__FlashAndSound_to_Flash_Transition_0;
      call_AcousticAlarm_SetOff)
  //
     mark_Region_0_conditional_consumed()
  //
    (
          transition_Unlocked_to_Locked_Transition_0 =
      [] /* other transition paths */
    )
  // dequeue()
od
```

4.4 Input / Output

There is no canonical form to express borders of a system and I/O across these borders in UML. For our purposes, we use a self-defined, minimal UML profile, providing the class stereotypes `<<system under test>>` and `<<environment>>` as we have already shown in Fig. 1. Classes without one of these stereotypes are considered to be part of the SUT.

There are several ways of communication between the SUT and its environment classes:

Incoming signals. In UML, incoming signals are modeled by signal receptions either in the SUT class or in an interface implemented by the SUT class. The latter is used for signals not directly handled by the SUT class but delegated to another class. In the OOAS code, this is modeled by a controllable action that puts the signal event into the event queues of all objects registered as listeners on this signal.

Outgoing signals. Outgoing signals are modeled as signal receptions in the environment classes in the UML model. In the OOAS code, this is mapped to an observable action that is called when the signal sending occurs.

Outgoing calls. In UML, outgoing calls are modeled as methods of environment classes (like the *setOn*/*setOff* methods in the Car Alarm System). In the OOAS this is mapped to a call of an observable action. If the callee has a return value, the observable action is directly followed by a controllable action.

Incoming calls. Incoming calls are modeled as method invocations of the SUT class in UML. In the OOAS this is reflected by a call of a controllable action. If the callee has a return value, the controllable action is directly followed by an observable action.

The classes in the OOAS directly derived from classes in the UML model are accompanied by two additional classes: *__model* and *__environment*. The *__model* class is put before the SUT class in a prioritized composition and provides house-keeping functionality like distributing broadcast signals and time triggers (see next subsection). The *__environment* class is put after the SUT class in a prioritized composition in the SAB and contains all the controllable actions like signal receptions. The system assembling block for the CAS is shown below.

```
/* all definitions before */
system
  __model // AlarmSystem // __environment
```

External inputs are put last in the OOAS execution because internal operations are assumed to happen in zero time steps and therefore would always be completed before the next input can happen - resulting in a run-to-completion behavior.

One important semantical difference between the behavior of the OOAS- and the UML model concerns the handling of input events that do not enable any transition in the current state of the state-machine. In the UML standard for state-machines (this is a semantic variation point for protocol state-machines) it is requested that these inputs are ignored (cf. *[1], State Machine, p. 566*). We deviate from the behavior specified in the UML standard and only allow inputs that enable transitions. The reasons for this design choice are among the following.

- Firstly, we use the input-output conformance relation (IOCO [11]) for testing. This conformance relation enables us to work with partial models, which is an advantage we want to preserve. Allowing all input events at all times would make the testing model *input complete* and disallow the use of several partial models working on the same inputs for test-case generation. Notice that IOCO assumes the implementation to be input-enabled.
- Secondly, disallowing inputs that are ignored also has the benefit of shrinking the state-space, which is an advantage during test-case generation: models of other demonstrators within the MOGENTES project are significantly more complex than the car alarm system.

Hence, we limit the OOAS to a non-input-enabled system. Within the OOAS-code, enabling and disabling of controllables is controlled via flags that are managed by the *__model* class.

4.5 Time Triggers

A time triggered transition fires a given amount of time after entering the source state if the state has not been left before. Object-oriented action systems, as described, have no notion of time, hence we need to emulate it. We also need to say that our support of time is restricted to cases of observable, variable delay: we do not control the SUT via timeouts.

We use an additional action *after(t)*, which is non-deterministically composed with the actions representing input from the environment, to mark the observation of passing time. Notice that we do not allow the waiting time to be split: two consecutive *after(t)* may not occur without either the first causing a time trigger to fire or a controllable action is used between them. This avoids series of *after(t)* actions which can be represented by one having a larger t parameter.

The time trigger functionality is realized by managing an ordered list of active timers. When a state with a leaving time triggered transition is entered, the timer is registered with the value of the time trigger.

The occurrence of *after(t)* reduces all timers in the list by the value of t. When a timer value is reduced to zero this way, the corresponding time trigger event is added to the event queue of the registered object. To simplify the implementation, we limit the allowed value of t to the minimum value of all registered timers. The following pseudo-code sketches the after action.

```
obs after (c_waittime : t_time) =
  requires c_waittime > 0 and wait_allowed and
      (len m.get_timers() > 0) and (t = min_timeout(m.get_timers())):
    /*update timers and event queue*/
end

do
```

```
    var t : t_time : after(t)
  []
    receive_external_signal_Close()
    /* further receptions */
od
```

Always taking the minimum time of the next timer due as a waiting time disallows certain behaviors, as is demonstrated in Fig. 4.

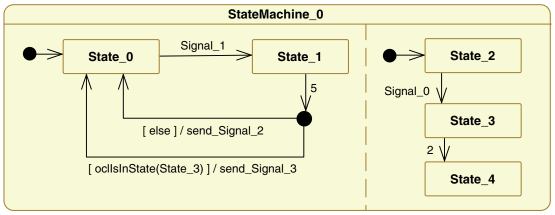

Fig. 4. Simple State Machine Example with Interdependent Time Triggers

In the example, due to the restriction to move to the next timer firing, a trace like

```
ctr Signal_1, obs after(4), ctr Signal_0 ,after(1), obs Signal_3
```

cannot be taken any longer. Therefore we lose the observation *Signal_3* in the example. In order to mitigate this shortcoming without the need to enumerate every time value that is smaller than the time value of the next timer firing, we propose to add non-deterministic transitions in the UML model that explicate these additional interleavings. Notice that this might be done on the fly by a tool that over-approximates such situations and adds the relevant transitions.

5 Results

We have implemented the presented transformations in a tool chain comprising two applications. The first utility takes a UML model and generates the object-oriented action system code. The second tool ("Argos") then converts the OOAS to an action system that is the input for our test-case generator called "Ulysses" [12,13]. As a second option, Argos is able to generate an implicit labeled-transition system in CADP-style (cf. [14]). All figures in this section were produced using this second back-end of Argos and the CADP tools.

Table 3 shows some basic figures for three different models of the car alarm system. The two one-region-only models are branching bisimilar [15] and included (modulo branching equivalence) in the multi-region model. Within the table, *B-Min.* stands for a minimization using strong bisimulation [16], while *W-Min.* stands for a sequence of weak-trace minimization that eliminates all

Table 3. Comparison of CAS Models

		One Region		Mult. Regions
LoC UML	[#]	–	263	273
LoC OOAS	[#]	105	720	790
LoC impl. LTS	[#]	2880	13 420	14 430
States	[#]	28	167	503
B-Min. States	[#]	22	58	159
W-Min. States	[#]	18	18	26
Transitions (hidden)	[#]	37 (7)	202 (140)	742 (662)
B-Min. Transitions (hidden)	[#]	31 (4)	67 (39)	178 (134)
W-Min. Transitions	[#]	27	27	42
Time Gen./Compile	[sec]	0.1/1.0	0.3/2.5	0.4/2.5

internal transitions followed by a minimization using strong bisimulation. The lines-of-code (LoC) figures as well as the time figures are reproduced only in order to give a hint on the complexity and performance. The experiments were carried out within a virtual machine running Ubuntu 9.04 on a Lenovo T400 laptop running Windows Vista. During the development of the models (and tools), we extensively used the animation, model-checking, minimization, and bisimulation capabilities offered by the CADP toolbox.

The first (left-to-right) model in the table is a hand-crafted OOAS model of our one-region car alarm system. It serves the purpose of showing the minimal number of lines necessary to model the CAS behavior. The second model is the UML-model presented in Fig. 2, and the third one is the multiple-regions CAS-model that was presented in Fig. 3.

Because the first model omits any event-handling overhead it has significantly less lines of code than the second model. It can also be seen that the percentage of hidden, i.e. internal, transitions is much lower for the first model: the event processing machinery contributes a lot of hidden transitions. These additional transitions also blow up the non-minimized state space, as can be seen in the table (167 vs. 28 states). Notice that the third model defines more testable behavior, which is reflected in additional states and transitions. In particular the third model allows for tests that send *Open* and *Close* events while being in the alarm state. However, it can be proved that under branching bisimulation (we are not interested in internal transitions) all the behavior specified in the second model (Fig. 2) is still present in the third one (Fig. 3).

While for the first model only one object is instantiated, models two and three comprise three objects each: the first object is used for event processing and house keeping, the second one models the alarm system itself, and the last one models the environment and it's capabilities of sending events.

Finally, Fig. 5 shows the explored state-space (weak-trace minimized) of our running one-region example. Notice the appearance of the observable "after", that models the observation of passing time.

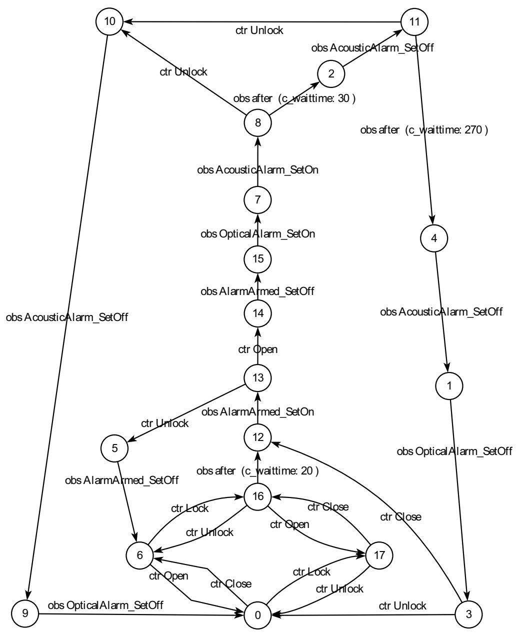

Fig. 5. Alarm System - Labeled Transition System

6 Conclusion

We have presented our mapping of a UML-subset to object-oriented action systems. It turns out that the mapping is relatively straight forward. In particular, we map concurrency to standard conformant non-deterministic choice, treat event processing as in-order and loss-less, and support time triggered transitions via timer queues. Having said that, some of our design decisions give our models a behavior that deviates from the UML standard: In case of time triggered transitions, we have proposed a way around this limitation, while in the case of the non-input-enabledness of the model we argue with the support of partial test-models. It is important to say that none of these choices constitute a principal limitation of our approach.

Other contributions of this paper are the extension of object-oriented action systems with prioritized composition and a system assembling block, the presentation of a tool chain that maps UML diagrams to labeled transition systems, and the discussion of a case study taken from industry. We have also demonstrated our ability to check that a refined model preserves the behavior of the more abstract one and we have given hints on how we validate our tools.

It is out of the scope of this paper to review all UML semantics, however, closest to our work on mapping UML to action systems is work on defining a UML profile for action systems (cf. [17]). This work is exactly the opposite of ours, as it aims to add a special UML profile that maps one-to-one to action systems. There has also been work on defining a mapping of UML to B which, according to [18], did not entirely meet the expectations as *schematic translations that attempt to cover a broad class of UML models usually result in B models that are hard to read and quite unnatural*. Because we do not aim at supporting a broad class of UML models in MOGENTES – in fact we are interested in supporting (partial) test models that are made from the requirements – and since the mapping to object-oriented action systems feels very natural, we do not suffer from the problem of 'unnatural' OOAS models. (Automatically generated code, however, always is a pity to read.)

By giving the action systems abstract trace semantics and generating labeled transition systems for them, we can leverage existing tools, such as the well-known CADP toolbox: checking of model-inclusion, absence of particular properties, and test-case generation becomes the problem of invoking the right CADP tool.

Finally, future work will concentrate on dealing with more complex models and finishing tool support for inheritance.

References

1. OMG: OMG Unified Modeling Language (OMG UML), superstructure, Version 2.2. (2009)
2. Back, R.J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. Distributed Computing 3(2), 73–87 (1989); Appeared previously in 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (1983)
3. Back, R.J., Sere, K.: Stepwise refinement of action systems. Structured Programming 12, 17–30 (1991)
4. Bonsangue, M.M., Kok, J.N., Sere, K.: An approach to object-orientation in action systems. In: Jeuring, J. (ed.) MPC 1998. LNCS, vol. 1422, pp. 68–95. Springer, Heidelberg (1998)
5. Sekerinski, E., Sere, K.: A theory of prioritizing composition. Technical Report 5, Turku Centre for Computer Science (1996)
6. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Inc., Englewood Cliffs (1976)
7. Back, R.J., Sere, K.: Superposition refinement of parallel algorithms. In: Proceedings of the IFIP TC6/WG6.1 Fourth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE 1991, pp. 475–493. North-Holland Publishing Co, Amsterdam (1992)
8. Fitzgerald, J., Larsen, P.G.: Modelling systems: practical tools and techniques in software development. Cambridge University Press, New York (1998)
9. Lucas, P.: Formal semantics of programming languages: VDL. IBM J. Res. Dev. 25(5), 549–561 (1981)
10. Butler, M., Morgan, C.: Action systems, unbounded nondeterminism, and infinite traces. Formal Aspects of Computing 7, 37–53 (1995)
11. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
12. Brandl, H., Weiglhofer, M., Aichernig, B.K.: Automated conformance verification of hybrid systems. In: QSIC (2010) (under review)
13. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Model-based mutation testing of hybrid systems. In: Proceedings of Formal Methods for Components and Objects FMCO 2009 (2010) (under review)

14. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
15. Glabbeek, R.v., Weijland, W.: Branching time and abstraction in bisimulation semantics (extended abstract). In Ritter, G., ed.: Information Processing 89, Proceedings of the IFIP 11th World Computer Congress, San Fransisco 1989, North-Holland (1989) 613–618 Full version in *Jounal of the ACM* 43(3), 1996, pp. 555–600.
16. Park, D.: Concurrency and automata on infinite sequences. In: Proceedings of the 5th GI-Conference on Theoretical Computer Science, London, UK, pp. 167–183. Springer, Heidelberg (1981)
17. Westerlund, T., Seceleanu, T.: An UML profile for action systems. Technical Report 581, Turku Centre for Computing Science (December 2003)
18. Fekih, H., Ayed, L.J.B., Merz, S.: Transformation of B specifications into UML class diagrams and state machines. In: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006, pp. 1840–1844. ACM, New York (2006)