Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking

Bernhard K. Aichernig Elisabeth Jöbstl
Institute for Software Technology
Graz University of Technology
Graz, Austria
aichernig@ist.tugraz.at joebstl@ist.tugraz.at

Model-based mutation testing uses altered test models to derive test cases that are able to reveal whether a modelled fault has been implemented. This requires conformance checking between the original and the mutated model. This paper presents an approach for symbolic conformance checking of action systems, which are well-suited to specify reactive systems. We also consider non-determinism in our models. Hence, we do not check for equivalence, but for refinement. We encode the transition relation as well as the conformance relation as a constraint satisfaction problem and use a constraint solver in our reachability and refinement checking algorithms. Explicit conformance checking techniques often face state space explosion. First experimental evaluations show that our approach has potential to outperform explicit conformance checkers.

1 Introduction

In most cases, full verification of a piece of software is not feasible. Possible reasons are the increasing complexity of software systems, the lack of highly-educated staff or monetary restrictions. In order to ensure quality and validate system requirements, testing is a viable alternative if it is systematic and automated. Model-based testing fulfills these criteria. The test engineer creates a formal model that describes the expected behaviour of the system under test (SUT). Test cases are then (automatically) derived from this test model by applying different algorithms and test specifications.

One big question is where to get the test specifications from. Our approach is fault-centred, i.e., mutation-based. Classical mutation testing is a method to assess and increase the quality of an existing test suite. The source code of the original program is syntactically altered by applying patterns of typical programming errors, so-called *mutation operators* [14, 15]. The test cases are then executed on the generated *mutants*. If not at least one test case is able to kill a mutant, the test suite has to be improved. Mutation testing relies on two assumptions that have been empirically confirmed: (1) The *competent programmer hypothesis* states that programmers are skilled and do not completely wrong. It assumes that they only make small mistakes. (2) The *coupling effect* states that test cases which are able to detect simple faults (like faults introduced by mutations) are also able to reveal more complex errors.

We employ the mutation concept on the test model instead of the source code and generate test cases that are able to kill the mutated models (*model-based mutation testing*). The generated tests are then run on the SUT and will detect whether a modelled fault has been implemented. So far, much more effort has been spent on the definition of mutation operators and classical mutation testing and not so much work has been done on test case generation from mutations [17].

What we have not mentioned so far: It is possible that a mutant does not show any different behaviour from the original program, although it has been syntactically changed. In this case, the mutant is equivalent to the original and no test case exists that can distinguish the two programs. In general, it is not

A. Petrenko, H. Schlingloff (Eds.): Workshop on
Model-Based Testing 2012 (MBT 2012)
EPTCS 80, 2012, pp. 88–102, doi:10.4204/EPTCS.80.7

decidable whether two programs are equivalent. Hence, mutation testing and its wider application are constrained by the *equivalent mutants problem* [17]. For test case generation, we also have to tackle this problem. Only if the original and the mutated model are not equivalent, we can generate a distinguishing test case. In our case, we do not check for total equivalence, but for refinement. The models we use are *action systems*, which were originally introduced by Back [8]. Action systems are well-suited for modelling reactive systems and allow non-determinism.

Within the European project MOGENTES [1], our group already developed a test case generation tool named *Ulysses*. It is basically an *ioco* checker for action systems and performs an explicit forward search of the state spaces. *ioco* is the input-output conformance relation by Tretmans [22]. Ulysses does not only work for discrete systems, but also supports hybrid action systems via qualitative reasoning techniques [11]. Experiments have shown that the performance of explicit enumeration of the state space involves high memory consumption and runtimes when being applied on complex models. In this paper, we present an alternative approach to determine (non-)refinement between two action systems.

As already shown in [6, 19], constraint satisfaction problems can be used to encode conformance relations and generate test cases. Each of this works dealt with transformational systems, i.e., systems that are started and take some input, process the input by doing some computations and then return an output and stop again. As already mentioned, action systems are well-suited to model reactive systems, i.e., systems that are continuously interacting with their environment. This kind of systems bring up a new aspect: reachability. Hence, the main contribution of this paper is a symbolic approach for refinement checking of reactive systems via constraint solving techniques that avoids state space explosion. We use the predicative semantics of action systems to encode (1) the transition relation and (2) the conformance relation as a constraint satisfaction problem. The constraint system representing the transition relation is used for a reachability analysis like it is known from model checking. For each reached state, we test whether it fulfills the constraint system that represents the conformance relation, which is refinement.

The rest of this paper is structured as follows. The next section presents our running example, a car alarm system. Section 3 gives an overview of the syntax and semantics of action systems and introduces the conformance relation we use. Section 4 explains our approach for finding differences between two action systems. Afterwards, Section 5 presents some experimental data on the application of our implementation on the car alarm system. Subsequently, Section 6 deals with restrictions and mentions some of our plans for future work. Finally, Section 7 discusses related work and concludes the paper.

2 Running Example

In order to demonstrate the basic concepts of our approach, we use a simplified version of a car alarm system (CAS). The example is taken from Ford's automotive demonstrator within the MOGENTES project. The following requirements were specified and served as the basis for our model:

R1 - Arming. The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.

R2 - 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.

R3 - Deactivation. The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

[1] `http://www.mogentes.eu`

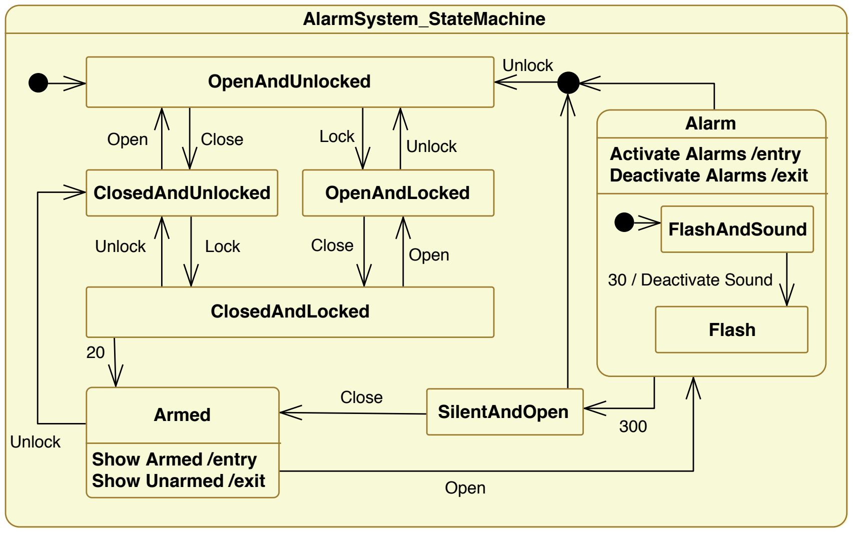

Figure 1: UML state machine of the car alarm system

Figure 1 shows a UML state machine of our CAS. From the state *OpenAndUnlocked* one can traverse to *ClosedAndLocked* by closing all doors and locking the car. As specified in requirement R1, 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. Similarly, 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. Note that the order of these two events is not specified, neither for enabling nor for disabling the alarms. Hence the system is not deterministic. When leaving the alarm state after a timeout (cf. requirement R2) 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 requirement R2, is reflected in the time-triggered transition leading to the *Flash* sub-state of the *Alarm* state.

3 Preliminaries

3.1 Action Systems

Action systems [8] are a kind of guarded-command language for modelling concurrent reactive systems. They have a formal semantics with refinement laws and are compositional [9]. Many extensions exist, but the main idea is that a system state is updated by guarded actions that may be enabled or not. If no action is enabled, the action system terminates. If several actions are enabled, one is chosen non-deterministically. Hence, concurrency is modelled in an interleaving semantics. The formal method *B* has recently adopted the action-system style in the form of *Event-B* [2].

Example 3.1. Our action systems are written in Prolog syntax. Listing 1 shows code snippets from the action system model of the CAS as described in Section 2. The first two lines contain user-defined types. All types are basically integers, but their ranges can be restricted. In Line 1, a type with name *enum_State* is defined. Its domain begins with 0 and ends with 7. Line 4 declares a variable with name *aState* which is of type *enum_State*. Line 6 defines the list of variables that make up the state of the action system.

Listing 1: Code snippet from the action system model for the car alarm system

```
 1 type(enum_State, X) :- X in 0..7.
 2 type(int, X) :- X in 0..270.
 3 ...
 4 var([aState], enum_State).
 5 ...
 6 state_def([aState, fromAlarm, fromArmed, ..., flashOn, soundOn]).
 7
 8 init([6, 0, 0, 0, 0, 0]).
 9
10 as :-
11     actions (
12     'after'(Wait_time)::(true) =>
13     (
14        ((Wait_time #= 20 #/\ aState #= 3) =>
15             (aState := 2; fromClosedAndLocked_OR_fromSilentAndOpen := 1))
16       []
17        ((Wait_time #= 30 #/\ aState #= 1 #/\ fromArmed #= 4) =>
18             (aState := 0; fromAlarm := 4; fromArmed := 0))
19       []
20        ((Wait_time #= 270 #/\ aState #= 0 #/\ fromAlarm #= 2) =>
21             (aState := 7; fromAlarm := 1; fromArmed := 0))
22     ),
23     'Lock'::(true) =>
24     (
25        ((aState #= 6 #/\ fromAlarm #= 0) => (aState := 5))
26       []
27        ((aState #= 4 #/\ fromArmed #\= 1) => (aState := 3; fromArmed := 0))
28     ),
29 ...
30     ),
31   dood (
32       'Lock'
33   [] [X:int]:'after'(X)
34   [] ...
35   ).
```

The *init* predicate in Line 8 defines the initial values for the state. At Line 10, the actual action system begins. It consists of an *actions* block (Lines 11 to 30) and an *do-od* block (Lines 31 to 35).

The *actions* block defines named actions. Each action consists of a name, a guard and a body (*name* :: *guard* => *body*) (cf. Lines 23 to 28). Actions may also have parameters, like action *after* in Line 12. The operator [] denotes non-deterministic choice. We use it in our example together with guards to distinguish between different cases in which an action may fire. Consider for example Lines 14 and 15. The action *after(20)* may fire if the action system is in a state where variable *aState* equals 3, which corresponds to state "ClosedAndLocked" in the CAS state chart (Figure 1). The action system then assigns variable *aState* value 2 and variable *fromClosedAndLocked OR fromSilentAndOpen* value 1, which corresponds to the state "Armed" in the state chart. The do-od block connects previously defined actions via non-deterministic choice. Basically, the execution of an action system is a continuous iteration over the do-od block. Here, there is always at least one action enabled. Hence, the car alarm system never terminates, but continuously waits for stimuli.

$$
\begin{array}{llll}
M & ::= D\ \texttt{as :-}\ \texttt{actions}(\overline{A}),\ \texttt{dood}(P). & P & ::= E \mid E\,[]\,P \\
D & ::= \overline{\texttt{type}(t,X)\ \texttt{:-}\ X\ \texttt{in}\ n_1..n_2.}\ \ \overline{\texttt{var}([\overline{v}],t).}\ \ \texttt{state\_def}([\overline{v}]).\ \texttt{init}([\overline{c}]). & E & ::= l \mid [\overline{X:t}]\,l(\overline{X}) \\
A & ::= L :: g => B & L & ::= l \mid l(\overline{X}) \\
B & ::= v := e \mid g => B \mid B;B \mid B\,[]\,B & e & ::= v \mid c \mid e+e \mid \ldots
\end{array}
$$

Figure 2: Syntax of a subset of action systems

$$
\begin{array}{llllll}
l :: g => B & =_{df} & g \wedge B \wedge tr' = tr^\frown[l] & l(\overline{X}) :: g => B & =_{df} & \exists\, \overline{X} : g \wedge B \wedge tr' = tr^\frown[l(\overline{X})] \\
x := e & =_{df} & x' = e \wedge y' = y \wedge \ldots \wedge z' = z & g => B & =_{df} & g \wedge B \\
B(\overline{v},\overline{v}');B(\overline{v},\overline{v}') & =_{df} & \exists\, \overline{v_0} : B(\overline{v},\overline{v_0}) \wedge B(\overline{v_0},\overline{v}') & B\,[]\,B & =_{df} & B \vee B
\end{array}
$$

Figure 3: Predicative semantics of actions

Syntax. In the literature many versions of Back's original action-system notation [8] exist. The syntax used in this work is presented in Figure 2. Our syntax contains some elements of Prolog, because the tool is implemented in SICStus Prolog. Here, an action system model M comprises the basic definitions D, a set of action definitions $\overline{A}$ and the do-od block P. In the basic definitions we define the types t, declare variables v of type t, define the system state-space as variable vector $\overline{v}$ and finally provide the initial state as vector of constants $\overline{c}$. An action A is a labelled guarded command with label L, guard g and body B. Actions may have a list of parameters $\overline{X}$. The body of an action may assign an expression e to a variable v or it may be composed of (nested) guarded commands itself. Composition may be sequential or non-deterministic choice. The do-od block P provides the event-based view on the action system. Here, the actions are composed by their action labels l. Currently, we only support non-deterministic choice in the do-od block, but in future sequential and prioritized composition will be added.

Semantics. The formal semantics of action systems is usually defined in terms of weakest preconditions. However, for our constraint-based approach, we found a relational predicative semantics being more suitable. We follow the style of He and Hoare's Unifying Theories of Programming [16]. Figure 3 presents the formal semantics of the actions of our modelling language. The state-changes of actions are defined via predicates relating the pre-state of variables $\overline{v}$ and their post-state $\overline{v}'$. Furthermore, the labels form a visible trace of events tr that is updated to tr' whenever an action runs through. Hence, a guarded action's transition relation is defined as the conjunction of its guard g, the body of the action B and the adding of the action label l to the previously observed trace. In case of parameters $\overline{X}$, these are added as local variables to the predicate. An assignment updates one variable x with the value of an expression e and leaves the rest unchanged. Sequential composition is standard: there must exist an intermediate state $\overline{v_0}$ that can be reached from the first body predicate and from which the second body predicate can lead to its final state. Finally, non-deterministic choice is defined as disjunction. The semantics of the do-od block is as follows: while actions are enabled in the current state, one of the enabled actions is chosen non-deterministically and executed. An action is enabled in a state if it can run through, i.e. if a post-state exists such that the semantic predicate can be satisfied. The action system terminates if no action is enabled. The labelling of actions is non-standard and has been added in order to support an event-view for testing.

3.2 Conformance

Once the modelling language with a precise semantics is fixed, we can define what it means that a SUT conforms to a given reference model, i.e. if the observations of a SUT confirm the theory induced by a formal model. This relation between a model and the SUT is called the conformance relation.

In model-based mutation testing, the conformance relation plays an additional role. It defines if a syntactic change in a mutant represents an observable fault, i.e. if a mutant is equivalent or not. However, for non-deterministic models an equivalence relation is no suitable conformance relation. An abstract non-deterministic model may do more than its concrete counterpart. Hence, useful conformance relations are order-relations rather than equivalence relations, the order going from abstract to more concrete models. In this work, we have chosen UTP's refinement relation as a conformance relation. UTP defines refinement via implication, i.e. more concrete implementations I imply more abstract models M.

Definition 3.1. ***(Refinement)***

$$M \sqsubseteq I \;=_{df}\; \forall x, x'y, y', \dots \in \alpha \;:\; I \Rightarrow M \qquad \textit{for all } M, I \textit{ with alphabet } \alpha.$$

The alphabet α is the set of variables denoting observations.

In [4] we have developed a mutation testing theory based on this notion of refinement. The key idea is to find test cases whenever a mutated model M^M does not refine an original model M^O, i.e. if $M^O \not\sqsubseteq M^M$. Hence, we are interested in counter-examples to refinement. From Definition 3.1 follows that such counter-examples exist if and only if implication does not hold:

$$\exists x, x', y, y', \dots \in \alpha \;:\; M^M \wedge \neg M^O$$

This formula expresses that there are observations in the mutant M^M that are not allowed by the original model M^O. We call a state, i.e. a valuation of all variables, *unsafe* if such an observation can be made.

Definition 3.2. ***(Unsafe State)*** *A pre-state u is called unsafe if it shows wrong (not conforming) behaviour in a mutated model M^M with respect to an original model M^O. Formally, we have:*

$$u \in \{s \mid \exists\, s' : M^M(s, s') \wedge \neg M^O(s, s')\}$$

We see that an unsafe state can lead to an incorrect next state. In model-based mutation testing, we are interested in generating test cases that cover such unsafe states. Hence, our fault-based testing criteria are based on the notion of unsafe states. How to search for unsafe states in action systems efficiently is discussed in the next section.

4 Searching Unsafe States

Figure 4 gives an overview of our approach to find an unsafe state. The inputs are the original action system model AS^O and a mutated version AS^M. Each action system consists of a set of actions AS_i^O and AS_j^M respectively, which are connected via non-deterministic choice. The first step is a preprocessing activity to check for refinement quickly. It is depicted on the left-hand side of Figure 4 as box *find mutated action*. If there does not exist an unsafe state at this point, we cannot find any mutated action that yields non-conformance. Hence, we already know that the action systems are equivalent. If we find an unsafe state in this phase, we cannot be sure that it is reachable from the initial state of the action system. But we know which action has been mutated and are able to construct a *non-refinement constraint*, which describes the set of all unsafe states. The next step performs a reachability analysis and uses the non-refinement constraint to test each reached state whether it is an unsafe state. In the following, we give more details.

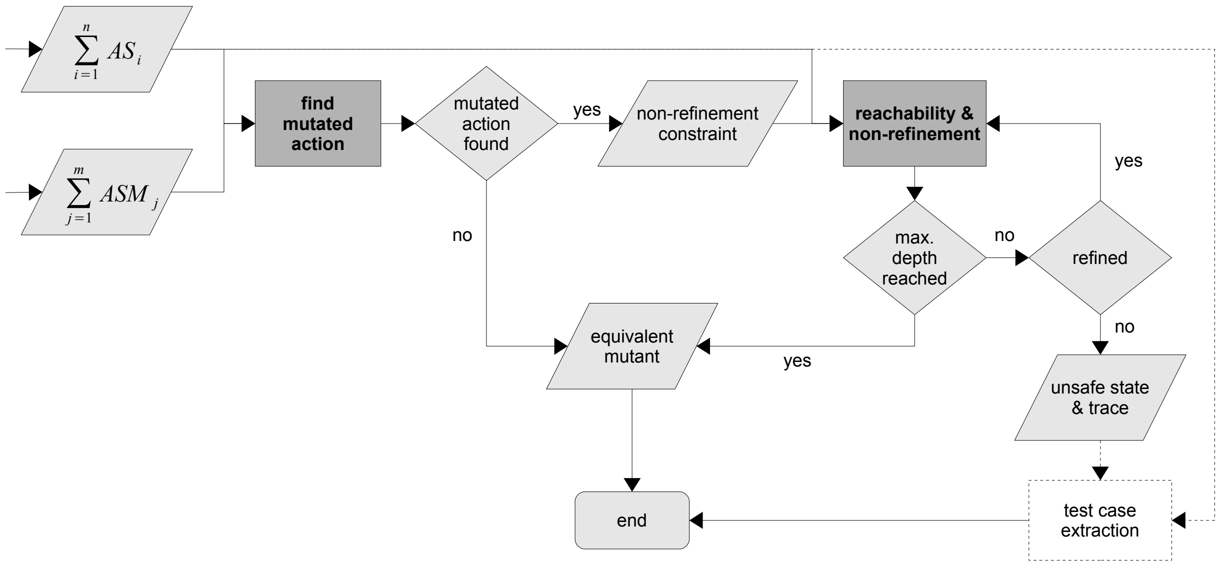

Figure 4: Process for finding an unsafe state

4.1 Non-Refinement of Action Systems

In the previous section, we have introduced non-refinement as a general criterion for identifying unsafe states. Now, we are going to concentrate on the special case of action systems.

The observations in our action system language are the event-traces and the system states before $(\overline{v}, tr)$ and after one execution $(\overline{v}', tr')$ of the do-od block. Then, a mutated action system AS^M refines its original version AS^O if and only if all observations possible in the mutant are allowed by the original. Hence, our notion of refinement is based on both, event traces and states. However, in an action system not all states are reachable from the initial state. Therefore, reachability has to be taken into account.

We reduce the general refinement problem of action systems to a step-wise simulation problem only considering the execution of the do-od block from reachable states:

Definition 4.1. ***(Refinement of Action Systems)*** *Let AS^O and AS^M be two action systems with corresponding do-od blocks P^O and P^M. Furthermore, we assume a function "reachable" that returns the set of reachable states for a given trace in an action system. Then*

$$AS^O \sqsubseteq AS^M =_{df} \forall \overline{v}, \overline{v}', tr, tr' : ((\overline{v} \in reachable(AS^O, tr) \wedge P^M) \Rightarrow P^O) \quad .$$

This definition is different to Back's original refinement definition based on state traces[9]. Here, also the possible event traces are taken into account. Hence, also the action labels have to be refined.

Negating this refinement definition and considering the fact that the do-od block is a non-deterministic choice of actions A_i leads to the non-refinement condition for two action systems:

$$\exists \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reachable(AS^O, tr) \wedge (A_1^M \vee \cdots \vee A_n^M) \wedge \neg A_1^O \wedge \cdots \wedge \neg A_m^O)$$

By applying the distributive law, we bring the disjunction outwards and obtain a set of constraints for detecting non-refinement.

Algorithm 1 *findMutatedAction*$(AS^O, AS^M) : (AS_i^M, CS\_nonrefine)$

```
1: $CS\_AS^O := trans(AS^O)$
2: for all $A_i^M \in AS^M$ do
3:     $CS\_AS_i^M := trans(A_i^M)$
4:     $CS\_nonrefine := CS\_AS_i^M \wedge \neg CS\_AS^O$
5:     if $sat(CS\_nonrefine)$ then
6:         return $(A_i^M, CS\_nonrefine)$      // mutated action found
7:     end if
8: end for
9: return $(nil, false)$      // equiv
```

Theorem 4.1. ***(Non-refinement)*** *A mutated action system AS^M does not refine its original AS^O, iff any action A_i^M of the mutant shows trace or state-behaviour that is not possible in the original action system:*

$$AS^O \not\sqsubseteq AS^M \quad \textbf{\textit{iff}} \quad \bigvee_{i=1}^{n} \exists \bar{v}, \bar{v}', tr, tr' \ : \ (\bar{v} \in reachable(AS^O, tr) \wedge A_i^M \wedge \neg A_1^O \wedge \cdots \wedge \neg A_m^O)$$

In the following, we discuss how this property is applied in our refinement checking process.

4.2 Finding a Mutated Action

The non-refinement condition presented in Theorem 4.1 is a disjunction of constraints of which each deals with one action A_i^M of the mutated action system AS^M. Hence, it is sufficient to satisfy one of these sub-constraints in order to find non-conformance. We use this for our implementation as we perform the non-refinement check action by action. Here, we first concentrate on finding a possibly unreachable unsafe state. Reachability is dealt with separately (see Section 4.3).

Algorithm 1 gives details on the action-wise non-refinement check, which is depicted on the left-hand side of Figure 4 (box *find mutated action*). We transform the whole do-od block of the original into a constraint system according to our predicative semantics of action systems (Line 1). We then translate one action of the mutated action system into a constraint system (Line 3). The non-refinement constraint *CS_nonrefine* is the conjunction of the constraint system representing the mutated action ($CS\_AS_i^M$) and the negated constraint system representing the original action system ($\neg CS\_AS^O$, cf. Line 4). Note that sequential composition involves existential quantification, which becomes universal quantification due to negation. Existential quantification is implicit in constraint systems. Universal quantification would lead to quantified constraint satisfaction problems (QCSPs) that are not supported by common constraint solvers. Fortunately, we can resolve this problem by a normal form that requires that non-deterministic choice is always the outermost operator and not allowed in nested expressions. In this way, the left-hand side of a sequential composition is always deterministic and existential quantification can be eliminated. Our car alarm system example (cf. Listing 1) already satisfies this normal form. Otherwise, each action system can be automatically rewritten to this normal form. This has not yet been implemented.

The non-refinement constraint for the just translated action is then given to a constraint solver to check whether it is satisfiable by any $\bar{v}, \bar{v}', tr, tr'$(Line 5), i.e., whether there exists an unsafe state $\bar{v}$ for AS^M and AS^O. If yes, we found the mutated action and return it together with the according non-refinement constraint *CS_nonrefine*. Otherwise, the next action A_i^M is investigated (loop in Line 2). If no action leads to a satisfiable non-refinement constraint, then AS^M refines AS^O (Line 9). Algorithm 1 is sound for first order mutants (one syntactical change per mutant). It aborts after finding the first action that leads to an unsafe state. Note that we do not know yet whether an unsafe state is actually reachable. For higher-order mutants (more than one syntactical change per mutant) it could happen that our algorithm finds a mutated action for which no unsafe state is reachable. In this case, it is necessary to go back and search for another mutated action until an unsafe state is actually reachable or all actions are processed.

Identifying the mutated action is important for our performance for two reasons: (1) Solving the non-refinement constraint *CS_nonrefine* for one action is by far faster than solving a non-refinement constraint encoding all actions of the mutated action system at once. Experiments showed that the latter is impractical with the currently used constraint solver. (2) By knowing which action has been mutated, we know which non-conformance constraint has to be fulfilled by an unsafe state. This saves constraint solver calls during the reachability analysis, which is presented in the following.

4.3 Reaching an Unsafe State

Now we know whether there exists any unsafe state. If this is the case, we also know which action has been mutated and we have determined a non-refinement constraint that describes the set of all possible unsafe states. But we do not know yet, whether an unsafe state is actually reachable from a given initial state. It is possible that an unsafe state exists theoretically and has been found in the previous step, but that no unsafe state is reachable from the initial state of the system. In this case, the mutated action system conforms to the original, i.e., the mutant refines the specification. To find out whether an unsafe state is actually reachable, we perform a state space exploration of the original action system AS. During this reachability analysis, each encountered state is examined if it is an unsafe state. This test is realized via a constraint solver that checks whether the reached state fulfills our non-refinement constraint (see right-hand side of Figure 4).

The pseudo-code in Algorithm 2 gives more details on our combined reachability and non-refinement check. The algorithm requires the following inputs: (1) the original action system AS^O, (2) the constraint system *CS_nonrefine* representing the non-refinement constraint obtained from Algorithm 1, (3) an integer *max* restricting the search depth, and (4) the initial state *init* of the action system AS^O. The algorithm returns a pair consisting of the found unsafe state and the trace leading there.

At first (Lines 1 to 3), we check whether the initial state is already an unsafe state. This is, we call the constraint solver with the non-refinement constraint and set the input state to be the initial state of AS. If the solver finds an action a leading to a post-state s then we detected non-conformance. We found either a state that can be reached from *init* only in the mutant but not in the original or an action that is enabled at state *init* only in the mutant but not in the original. In this case, *init* is returned as unsafe state together with the empty trace. Otherwise, we perform a breadth-first search (Lines 4 to 19) starting at *init*. The queue *ToExplore* holds the states that have been reached so far and still have to be further explored. It contains pairs consisting of the state and the shortest trace leading to this state. The set $Visited$ holds all states that have been reached so far and is maintained to avoid the re-exploration of states. To ensure termination, the state space is only explored up to a user-defined depth *max* (Line 9).

The function $succStateAndAction(s_0)$ (Line 10) returns the set of all successors of state s_0. Each successor is a pair consisting of the successor state s_1 and the action a_1 leading from s_0 to s_1. The successors are calculated via the predicative semantics of our action systems (cf. Section 3.1). Thereby, we gain a constraint system representing the transition relation of our action system. It describes one iteration of the do-od block. The interesting variables in the constraint system are the input state variables, the action variable, and the post-state variables. The input state variables are set to be equal to the variables in s_0.

Algorithm 2 $reachNonRefine(AS^O, CS\_nonrefine, max, init) : (unsafe, trace)$

```
1:  if $\exists a,s : CS\_nonrefine(init,a,s)$ then
2:      return $(init,[])$
3:  end if
4:  $Visited := \{init\}$
5:  $ToExplore := enqueue((init,[]),[])$
6:  while $ToExplore \neq []$ do
7:      $(s_0, tr\_s_0) := head(ToExplore)$
8:      $ToExplore := dequeue(ToExplore)$
9:      if $length(tr\_s_0) < max$ then
10:         for all $(s_1,a_1) \in succStateAndAction(s_0) : s_1 \notin Visited$ do
11:             $tr\_s_1 := add(tr\_s_0, a_1)$
12:             if $\exists a_2, s_2 : CS\_nonrefine(s_1,a_2,s_2)$ then
13:                 return $(s_1, tr\_s_1)$    // unsafe state
14:             end if
15:             $Visited := add(s_1, Visited)$
16:             $ToExplore := enqueue((s_1,tr\_s_1), ToExplore)$
17:         end for
18:     end if
19: end while
20: return $(nil,[])$    // equiv
```

We then use a constraint solver to set the action variable a_1 and the variables that make up the post-state s_1. By calling the constraint solver multiple times with an extended constraint system (with the added restriction that the next solution has to be different fromt the previous ones), we get all transitions that are possible from s_0.

Each state s_1 that is reached in this way and has not yet been processed ($s_1 \notin Visited$) is checked for being an unsafe state (Line 12). This works analogously to Line 1. If an unsafe state is found it is returned together with the trace leading there (Line 13). Otherwise, the state is included in the set of visited states (Line 15) and enqueued for further exploration (Line 16). If no unsafe state is found up to depth *max*, the mutant refines the original action system and we return the pair $(nil,[])$ as a result (Line 20).

4.4 Test Case Extraction

We implemented our technique in SICStus Prolog[2] (version 4.1.2). SICStus comes with an integrated constraint solver *clpfd* (Constraint Logic Programming over Finite Domains) [13], which we used. Our implementation results either in the verdict *equiv*, which means that the mutated action system conforms to the original, or in an unsafe state and a sequence of actions leading to this state. In the latter case it is possible to generate a test case. The trace resulting from our approach is not yet a test case, although it reaches the unsafe state. We still need to add verdicts (pass, fail, and inconclusive) where necessary. Additionally, the trace has to be at least one step longer in order to check that only correct behaviour occurs after the unsafe state. A test case generated in this way is able to reveal whether the model mutant has been implemented. This test case extraction step has not yet been implemented and remains future

[2]`http://www.sics.se/sicstus/`

work. It is indicated by the dotted parts at the right bottom of Figure 4. For an explicit *ioco* checking technique, we have suggested different test case extraction strategies in [3].

5 Empirical Results

For an empirical evaluation of our prototypical implementation, we have modelled the car alarm system (CAS) described in Section 2 as an action system. Some code snippets of the model have already been presented in Listing 1. Additionally, we have manually created first order mutants (one mutation per mutant) for the original CAS model. We applied the following three mutation operators:

- *guard true*: Setting all possible guards to true resulted in 34 mutants.
- *comparison operator inversion*: The action system contains two comparison operators: equality (`#=`) and inequality (`#\=`). Inverting all possible equality operators (resulting in inequality) yielded 52 mutants. Substituting inequality by equality operators resulted in 4 mutants.
- *increment integer constant*: Incrementation of all integer constants by 1 resulted in 116 mutants. Note that at the upper bound of a domain, we took the smallest possible value in order to avoid domain violations.

From these mutation operators, we obtained a total of 206 mutated action systems. Additionally, we also included the original action system as an equivalent mutant. Unfortunately, the currently used constraint solver was not able to handle 12 of the 207 mutants within a reasonable amount of time during refinement checking without reachability (see Section 4.2). We will try another constraint solver and see if the performance increases. For now we had to exclude the 12 mutants from our experiments.

We ran our experiments on a machine with a dual-core processor (2.8 GHz) and 8 GB RAM with a 64-bit operating system. Table 1 gives information about the execution times of our *refinement checker* prototype for the remaining 195 mutations. All values are given in seconds unless otherwise noted. We conducted our experiments for four different versions of the CAS: (1) **CAS_1**: the CAS as presented in Section 2 with parameter values 20, 30, and 270 for the action *after*, (2) **CAS_10**: the CAS with parameter values multiplied by 10 (200, 300, and 2700), (3) **CAS_100**: the CAS with parameters multiplied by 100, and (4) **CAS_1000**: the CAS with parameters multiplied by 1000. These extended parameter ranges shall test the capabilities of our symbolic approach. The column *find mutated action* shows that checking whether there possibly exists an unsafe state and which action has been mutated (see Section 4.2) is quite fast. The reachability and non-refinement check (column *reach & non-refine*, see Section 4.3) needs the bigger part of the overall execution time (column *total*). The four versions of the CAS differ only in the parameter values and the domains for the parameters. Our approach takes almost the same amount of time for all four versions: approximately 1¾ minutes to process all 195 mutants, on average half a second per mutant, a minimum time per mutant of 0.03 seconds, and a maximum of about 3 seconds for one mutant.

To have at least a weak reference point for our performance, we have also utilized our explicit *ioco* checker *Ulysses* [3, 11] to generate tests for the CAS. We have to admit that this comparison is not totally fair, since Ulysses works quite differently: First of all, Ulysses uses a different conformance relation named *ioco* (input-output conformance for labelled transition systems, see [22]). We ran Ulysses in two settings. First, on the CAS with distinguished input and output actions. The input actions were Close, Open, Lock, and Unlock. The remaining actions were classified as outputs. Second, we classified all actions of the CAS as outputs. This setting is closer to our notion of conformance, since in refinement we do not distinguish between input and output actions. Nevertheless, the conformance relations are still

CAS version		**refinement checker**			**Ulysses**	
		find mutated action	**reach & non-refine**	**total**	**in/out**	**out**
CAS_1	total	16	90	106	98	65
	average	0.08	0.46	0.54	0.50	0.34
	min.	0.01	0.02	0.03	0.05	0.05
	max.	0.30	2.80	3.10	6.30	5.33
CAS_10	total	15	86	101	8.8 h	7.9 h
	average	0.08	0.44	0.52	2.7 min	2.4 min
	min.	0.01	0.02	0.03	0.45	0.36
	max.	0.27	2.80	3.07	2.6 h	2.6 h
CAS_100	total	16	90	106	-	-
	average	0.08	0.46	0.54	-	-
	min.	0.01	0.02	0.03	-	-
	max.	0.27	2.77	3.04	-	-
CAS_1000	total	15	85	100	-	-
	average	0.08	0.44	0.52	-	-
	min.	0.01	0.02	0.03	-	-
	max.	0.27	2.69	2.96	-	-

Table 1: Execution times for our refinement checking tool and the *ioco* checker Ulysses applied on four versions of the car alarm system. All values are given in seconds unless otherwise noted.

not identical. In refinement, we only check that an implementation does not show unspecified behaviour. Hence, an implementation can always do less than specified. In *ioco*, abscence of (output) behaviour has to be explicitly permitted by the specification model. Another difference between Ulysses and our approach are the final results. Ulysses generates adaptive test cases, not only a trace leading to an unsafe state as our tool does (cf. Section 4.4).

Despite these inconsistencies, the comparison with Ulysses still demonstrates one thing very clearly: the problems with explicit state space exploration. Ulysses explicitly enumerates all symbolic values (like parameters in the CAS example). Table 1 also gives the execution times for Ulysses on the CAS with our two settings: (1) distinction between inputs and outputs (column *in/out*) and (2) every action is an output (column *out*). For the original CAS version (**CAS_1**), Ulysses is faster than our constraint-based approach, particularly if every action is an output. In this case, test case generation with Ulysses took only one minute for all 195 mutants. But when it comes to **CAS_10** with larger parameter values (200, 300, and 2700 instead of 20, 30, and 270) Ulysses runs into massive problems. The execution time drastically increases to almost 9 hours (in/out) and about 8 hours (out). On average, each mutant takes 2.7 to 2.4 minutes. One mutant even caused a runtime of 2.6 hours. We observed a memory usage of up to 6 GB RAM. We suspect that a significant amount of the execution time is spent on swapping. For the CAS versions **CAS_100** and **CAS_1000**, we did not run Ulysses as the runtimes would be even higher.

Already for the original CAS (**CAS_1**), Ulysses needs 5 to 6 seconds for some mutants that altered the *after* action that has one parameter: the time to wait with a range from 0 to 270. Our approach took only 0.1 seconds to find the unsafe state and the corresponding trace. Hence, Ulysses shows very good performance for systems with small domains. When it comes to larger ranges of integers, Ulysses comes to its limits quite soon. In this cases, our approach represents a viable alternative.

6 Restrictions and Further Optimizations

Although our approach shows great promise for solving the problems with large variable domains, it is far from being perfect. In the following, we discuss restrictions and possible optimizations of the overall approach as well as of our current implementation: More elaborate *conformance relations* are possible. In [23] we presented a predicative semantics for ioco. Alternating simulation is also an option.

As already discussed in Section 4.4, our approach currently results in an unsafe state and a trace leading there. The generation of *adaptive test cases* remains future work. Our action systems are ignorant of *time*. In the CAS the waiting time was modelled as a simple parameter. For more elaborate models with clocks a tick-action modelling the progress of time is needed. For a full timed-automata model, the actions could be extended with deadlines similar to [10].

One obvious improvement for our *implementation* is the use of more efficient data structures. Currently, we use lists in most cases as they are the most common data structure in Prolog. For example, the set of visited states in Algorithm 2 is currently represented by a list. The use of ordered sets in combination with hash values would be reasonable. As already mentioned, we implemented our approach in SICStus Prolog. It comes with a built-in constraint solver (*clpfd* - Constraint Logic Programming over Finite Domains [13]), which we use so far. Our next steps will include a comparison with other constraint solvers, e.g., Minion[3]. Additionally, we already supervise an ongoing diploma thesis on the use of different SMT solvers like Yices[4] or Z3[5].

7 Conclusion

This paper deals with model-based mutation testing. Like in classical model-based testing, we have a test model describing the expected behaviour of a system under test. This model is mutated by applying syntactical changes. We then generate test cases that are able to reveal whether a software system has implemented the modelled faults. We have chosen action systems as a formalism for system modelling. In this paper, we presented our syntax and a predicative semantics for action systems. We also explained refinement in the context of action systems. Most importantly, we have developed and implemented an approach for refinement checking of action systems as a first step for test case generation from mutated action systems. Throughout the whole paper, a car alarm system served as a running example, which was not only used for illustration but also served as a case study for our experiments.

We employ constraint satisfaction techniques that have already been used previously [6, 19] to encode conformance relations and generate test cases. Nevertheless, prior works dealt with systems that take an input and deliver some output. This paper deals with refinement checking of reactive systems. The thereby introduced continuous interaction with the environment brings up a new aspect: reachability. Hence, the main contribution of this paper is a symbolic approach for refinement checking of reactive systems via constraint solving techniques that avoids state space explosion, which is often a problem with explicit techniques.

Our approach to detect non-refinement in action systems is basically a combination of reachability and refinement checking. We use the predicative semantics of action systems to encode (1) the transition relation and (2) the conformance relation as a constraint satisfaction problem. During reachability analysis, the constraint system representing the transition relation is used for finding successor states. The

[3]http://minion.sourceforge.net
[4]http://yices.csl.sri.com/
[5]http://research.microsoft.com/en-us/um/redmond/projects/z3/

constraint system encoding refinement enables us to test each reached state whether it is an unsafe state, i.e., whether this state is directly followed by observations in the mutant that must not occur at this state according to the original model.

Experimental results with an action system modelling a car alarm system have demonstrated the potential of our approach compared to explicit conformance checking techniques. We conducted experiments with four different versions of the car alarm system that only differ in the integer ranges of the parameters. The smallest model deals with parameters from 0 to 270, the largest model contains integer parameters from 0 to 270000. Our implementation provides constant runtime for all four models. For 195 mutated models, we only need about 1¾ minutes regardless of the parameter ranges. The explicit conformance checker that we also applied on two model versions was faster (1 to 1½ minutes) for the smallest model, but already the next larger model caused an execution time of about 8 hours.

There is existing literature on model-based mutation testing. One of the first models to be mutated were predicate-calculus specifications [12] and formal Z specifications [21]. Later on, model checkers were available to check temporal formulae expressing equivalence between original and mutated models. In case of non-equivalence, this leads to counterexamples that serve as test cases [7]. This is very similar to our approach, but in contrast to this state-based equivalence test, we check for refinement allowing non-deterministic models. Another conformance relation capable to deal with non-determinism is the input-output conformance (*ioco*) of Tretmans [22]. The first use of an ioco checker for mutation testing was on LOTOS specifications [5]. The tool *Ulysses* that was already mentioned in Section 5 applies ioco checking for mutation-based test case generation on qualitative action systems [11]. A further conformance relation supporting non-determinism is FDR (Failures-Divergence Refinement) for the CSP process algebra [1]. The corresponding FDR model checker/refinement checker has been used in [20] to set up a whole testing theory in terms of CSP. This work allows test case generation via test purposes, but not by model mutation.

Our own past work has shown that typically there is no silver bullet in automatic test case generation that is able to deal with every system efficiently [18]. As we only used one exemplary model for evaluating our approach so far, it is too early to say whether the performance of our approach may be generalized. Future work will include more experiments with different types of systems to find this out.

Acknowledgment. Research herein was funded by the Austrian Research Promotion Agency (FFG), program line "Trust in IT Systems", project number 829583, TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation (TRUFAL).

References

[1] A. W. Roscoe (1994): *Model-checking CSP*, chapter 21. Prentice-Hall. Available at `http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/publications/50.ps`.

[2] Jean-Raymond Abrial (2010): *Modelling in Event-B: System and software design.* Cambridge University Press.

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl & Willibald Krenn (2011): *Efficient Mutation Killers in Action.* In: *IEEE 4th Int. Conf. on Software Testing, Verification and Validation, ICST 2011*, IEEE Computer Society, pp. 120–129. Available at `http://dx.doi.org/10.1109/ICST.2011.57`.

[4] Bernhard K. Aichernig & Jifeng He (2009): *Mutation testing in UTP. Formal Aspects of Computing* 21(1-2), pp. 33–64, doi:10.1007/s00165-008-0083-6.

[5] Bernhard K. Aichernig, Bernhard Peischl, Martin Weiglhofer & Franz Wotawa (2007): *Protocol Conformance Testing a SIP Registrar: An Industrial Application of Formal Methods.* In: *5th IEEE Int. Conf. on

Software Engineering and Formal Methods, SEFM 2007, IEEE Computer Society, pp. 215–224. Available at `http://doi.ieeecomputersociety.org/10.1109/SEFM.2007.31`.

[6] Bernhard K. Aichernig & Percy Antonio Pari Salas (2005): *Test Case Generation by OCL Mutation and Constraint Solving*. In: *5th Int. Conf. on Quality Software, QSIC 2005*, IEEE Computer Society, pp. 64–71. Available at `http://doi.ieeecomputersociety.org/10.1109/QSIC.2005.63`.

[7] Paul Ammann, Paul E. Black & William Majurski (1998): *Using Model Checking to Generate Tests from Specifications*. In: *2nd IEEE Int. Conf. on Formal Engineering Methods, ICFEM 1998*, IEEE Computer Society, pp. 46–54. Available at `http://computer.org/proceedings/icfem/9198/91980046abs.htm`.

[8] Ralph-Johan Back & Reino Kurki-Suonio (1983): *Decentralization of Process Nets with Centralized Control*. In: *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, ACM, pp. 131–142.

[9] Ralph-Johan Back & Kaisa Sere (1991): *Stepwise Refinement of Action Systems*. *Structured Programming* 12, pp. 17–30.

[10] Sébastien Bornot, Joseph Sifakis & Stavros Tripakis (1998): *Modeling Urgency in Timed Systems*. In: *Compositionality: The Significant Difference (COMPOS'97)*, *LNCS* 1536, Springer, pp. 103–129.

[11] Harald Brandl, Martin Weiglhofer & Bernhard K. Aichernig (2010): *Automated Conformance Verification of Hybrid Systems*. In: *10th Int. Conf. on Quality Software, QSIC 2010*, IEEE Computer Society, pp. 3–12. Available at `http://dx.doi.org/10.1109/QSIC.2010.53`.

[12] Timothy A. Budd & Ajet S. Gopal (1985): *Program testing by specification mutation*. *Computer languages* 10(1), pp. 63–73. Available at `http://dx.doi.org/10.1016/0096-0551(85)90011-6`.

[13] Mats Carlsson, Greger Ottosson & Björn Carlson (1997): *An Open-Ended Finite Domain Constraint Solver*. In: *9th Int. Symp. on Programming Languages: Implementations, Logics, and Programs*, PLILP '97, Springer, pp. 191–206. Available at `http://dl.acm.org/citation.cfm?id=646452.692956`.

[14] R. DeMillo, R. Lipton & F. Sayward (1978): *Hints on test data selection: Help for the practicing programmer*. *IEEE Computer Society* 11(4), pp. 34–41.

[15] Richard G. Hamlet (1977): *Testing programs with the aid of a compiler*. *IEEE Transactions on Software Engineering* 3(4), pp. 279–290.

[16] C.A.R. Hoare & Jifeng He (1998): *Unifying Theories of Programming*. Prentice-Hall International.

[17] Yue Jia & Mark Harman (2011): *An Analysis and Survey of the Development of Mutation Testing*. *IEEE Transactions on Software Engineering* 37(5), pp. 649–678. Available at `http://doi.ieeecomputersociety.org/10.1109/TSE.2010.62`.

[18] Elisabeth Jöbstl, Martin Weiglhofer, Bernhard K. Aichernig & Franz Wotawa (2010): *When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving*. In: *3rd Int. Conf. on Software Testing, Verification and Validation, ICST 2010*, IEEE Computer Society, pp. 479–488. Available at `http://doi.ieeecomputersociety.org/10.1109/ICST.2010.48`.

[19] Willibald Krenn & Bernhard K. Aichernig (2009): *Test Case Generation by Contract Mutation in Spec#*. *Electronic Notes in Theoretical Computer Science* 253(2), pp. 71 – 86, doi:10.1016/j.entcs.2009.09.052.

[20] Sidney Nogueira, Augusto Sampaio & Alexandre Mota (2008): *Guided Test Generation from CSP Models*. In: *5th Int. Colloquium on Theoretical Aspects of Computing, ICTAC 2008*, *LNCS* 5160, Springer, pp. 258–273. Available at `http://dx.doi.org/10.1007/978-3-540-85762-4_18`.

[21] Philip Alan Stocks (1993): *Applying formal methods to software testing*. Ph.D. thesis, Department of computer science, University of Queensland.

[22] Jan Tretmans (1996): *Test Generation with Inputs, Outputs and Repetitive Quiescence*. *Software - Concepts and Tools* 17(3), pp. 103–120.

[23] Martin Weiglhofer & Bernhard Aichernig (2010): *Unifying Input Output Conformance*. In: *Unifying Theories of Programming*, *LNCS* 5713, Springer, pp. 181–201. Available at `http://dx.doi.org/10.1007/978-3-642-14521-6_11`.