TECHNICAL REPORT

IST-MBT-2013-01

Fast Refinement Checking for Test Case Generation

Bernhard K. Aichernig, Elisabeth Jöbstl,
Matthias Kegele

{aichernig, joebstl}@ist.tugraz.at,
matthias.kegele@student.tugraz.at

January 2013

Institute for Software Technology (IST)
Graz University of Technology
A-8010 Graz, Austria

Fast Refinement Checking for Test Case Generation

Bernhard K. Aichernig, Elisabeth Jöbstl, and Matthias Kegele

Institute for Software Technology, Graz University of Technology, Austria
{aichernig,joebstl}@ist.tugraz.at, matthias.kegele@student.tugraz.at

Abstract. We combine model-based testing and mutation testing to automatically generate a test suite that achieves a high mutation adequacy score. The original model representing the system under test is mutated. To generate test cases that detect whether a modelled fault has been implemented, we perform a refinement check between the original and the mutated models. Action systems serve as formal models. They are well-suited to model reactive systems and allow non-determinism. We extend our previous work by two techniques to improve efficiency: (1) a strategy to efficiently handle a large number of mutants and (2) incremental solving. A case study illustrates the potential of our improvements. The runtime for checking approximately 200 mutants could be reduced from 20 to 3 seconds. We implemented our algorithms in two versions: one uses a constraint solver, the other one an SMT solver. Both show similar performance.

1 Introduction

We combine model-based testing [33] and classical mutation testing [24, 19] in order to automatically generate test cases that achieve a high mutation adequacy score. As pointed out by Jia and Harman in their recent survey on mutation testing [26], *"practical software test data generation for mutation test adequacy remains an unresolved problem"*. Furthermore, they identified a *"pressing need"* to address this problem.

Fig. 1 gives an overview of our approach. The left-hand side (non-grey parts) refers to model-based testing: it is assumed that the source code of the system under test (SUT) is not accessible (black-box testing). Therefore, a tester develops a formal model describing the expected behaviour of the SUT. This model is assumed to be correct with respect to some properties derived from the requirements. This can be assured, e.g. via model checking. It serves as input for our test case generation tool, where it is used to generate the input stimuli and as a test oracle for the expected behaviour. The resulting tests are automatically executed. If all tests pass, we have conformance between the model and the SUT. However, since exhaustive testing is impractical, we have to select a proper subset of possible test cases. We accomplish this by combining this model-based testing approach with mutation testing. As we have no access to the source code of the SUT, we mutate our model of the SUT (cf. grey parts in Fig. 1). Then,

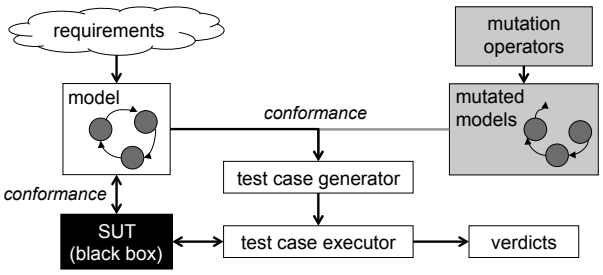

Fig. 1. Model-based mutation testing

given the original model and a set of mutated models, we automatically generate test cases that kill the model mutants. The generated tests are then executed on the SUT and will detect if a mutated model has been implemented. Hence, our model-based mutation testing approach is fault-centred. It tests rather against non-conformance than for conformance - we are rather aiming for falsification than for verification.

In our approach, equivalent model mutants are singled out automatically. In contrast to the original idea of program mutation [24, 19], where a given set of test cases is analysed, here we generate a test suite that will kill all (non-equivalent) model mutants. This is non-trivial, since it involves an equivalence (conformance) check between original and mutated models (cf. Fig. 1), which we focus on in this paper. Since, equivalence is undecidable in general, we restrict ourselves to bounded domains.

Though the problem of checking for equivalence of two systems is hard enough already, we do also allow non-determinism in our models. In a non-deterministic model, a given (sequence of) input stimuli may cause several possible output observations. Non-determinism arises due to abstraction that is frequently required in good test models. When comparing two non-deterministic models (an original and a mutant in our case), equivalence is not sufficient any more. Hence, the conformance relation needs to be an order or preorder relation. Refinement is such an order relation [20]. We implemented a refinement checker for non-deterministic models in order to enable the generation of test cases that are able to detect whether a faulty model has been implemented in the SUT.

In principle, this model-based mutation testing approach could be implemented with existing tools, like model checkers [21]. We refrained from that as model checkers are more general and therefore more complex than necessary. Thus, adaptions to optimise existing model checkers for our particular needs would probably not be beneficial. Furthermore, for generating test cases we need access to the internals of the state space. The counterexample trace is insufficient for non-deterministic models.

Previous Work. A first implementation of our model-based mutation testing approach has been implemented in our tool *Ulysses*. It is basically checking input-output conformance [32] of two action systems and performs an explicit forward search of the state spaces. Our experience with the tool shows that the

performance of explicit enumeration of the state space involves high memory consumption and runtimes. In [5] we illustrated that this is even the case for rather small models involving parameters. To overcome this and allow the handling of complex models, we work on an efficient tool using symbolic handling of data. We already presented some aspects of our work previously: we explained the basic refinement checking approach [5], pointed out problems that need to be handled [6], and presented techniques that considerably improved the efficiency of our approach [4]. The main contributions of this paper are two further improvements on efficiency: the efficient handling of a large number of mutants and the implementation of incremental solving in constraint logic programming. We implemented our approach and our improvements in Prolog using a constraint solving library. We conducted a first case study on a car alarm system that shows the benefits of our improvements: the execution time could be reduced from approximately 20 to about 3 seconds. This is a reduction of about 85%. In addition, we also tried an implementation using SMT solving for our case study, but could not identify a further performance gain.

The rest of this paper is organised as follows: In Section 2, we introduce preliminaries, i.e., our modelling language and our notion of refinement. In Section 3, we explain our refinement checking approach and present two techniques to increase its efficiency in Section 4. Their implementations are discussed in Section 5. In Section 6, we report on our case study. In Section 7, we cover related work and we conclude in Section 8.

2 Preliminaries

2.1 Action Systems

Our chosen modelling formalism are action systems [9], which are well-suited to model reactive and concurrent systems [10]. They have a formal semantics with refinement laws and are compositional [11]. Many extensions exist, e.g. object-oriented action systems [14], 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 nondeterministically. Hence, concurrency is modelled in an interleaving semantics.

Syntax. There exist various versions of Back's original action system notation [9]. The syntax we use is defined as follows:

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

It contains some Prolog elements, since our refinement checking tool is implemented in Prolog. An action system model M consists of basic definitions D, action definitions $\overline{A}$, and a do-od block P. D comprises the definition of types t, the declaration of variables v of type t, the definition of the system state as a variable vector $\overline{v}$, and the definition of the initial state as a 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. Actions may be composed by sequential composition ; or non-deterministic choice []. The do-od block P provides the event-based view on the action system. It composes the actions by their action labels l via non-deterministic choice.

Example 1. The following code snippet is part of an action system modelling a car alarm system. It defines the action *'AlarmOn'*, which turns on the flash lights and the signal-horn in case an alarm has been triggered.

```
1  type(bool, X) :- X in 0..1.
2  var([f, s], bool).
3  state_def([f, s, ...]).
4  init([0, 0, ...]).
5  as :-
6    actions (
7       'AlarmOn'::(f #= 0 #/\ s #= 0) => (
8          ((f := 1 ; s := 1)    % ((f := 0) ; s := 1)
9           []
10           (s := 1 ; f := 1)),
11    ...),
12    dood ('AlarmOn' [] ... ).
```

All data types in our action systems are integers with restricted ranges. Line 1 defines the type 'bool' with two possible values: 0 or 1. Line 2 declares two variables with name f and s which are of type *bool*. They are used to indicate whether the flash lights and the sound (signal horn) are turned on. Line 3 defines the list of variables that make up the state of the action system. The initial values for the state are defined in Line 4. The *actions* block (Lines 6 to 11) defines named actions, which consist of a name, a guard and a body (*name* :: *guard* => *body*). The action *'AlarmOn'* (Lines 7 to 10) models the activation of the alarms. This is only possible if both alarms are turned off before (guard in Line 7). It is not specified in which order the two alarms are turned on. This is modelled by the non-deterministic choice ([]) of two sequential compositions (;). Either, Line 8 turns on the flash first and afterwards the sound or Line 11 first turns on the sound and then the flash. The *do-od* block (Line 13) connects previously defined actions via non-deterministic choice. Basically, the execution of an action system is a continuous iteration over the do-od block.

Our overall goal is to generate a test case that is able to detect certain faults. For this purpose, we mutate our models and perform a refinement check between the original and the mutated model. The comment (%) in Line 8 represents a possible mutation. It assigns the variable f the value 0 instead of 1. This leads

to a difference in the behaviour of the original action system (the specification) and the mutated one. The original activates two alarms (flash and sound), i.e., it sets the variables f and s to true. The mutated action system cannot always establish this behaviour. Although it results in the correct post-state by choosing Line 10, it might also end up in a wrong post-state by executing the mutated statement of Line 8. In this case, the flash will not be turned on. This scenario is a counterexample to refinement. It allows us to derive a test case that checks whether the modelled fault has been implemented in a SUT.

Semantics. The formal semantics of action systems is typically defined in terms of weakest preconditions. However, we found a relational predicative semantics being more suitable for our constraint-based approach. In [6], we gave reasons for our choice. Our formal semantics of actions is defined as follows:

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

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.

Example 2. The predicative semantics of the action *'AlarmOn'* in Example 1 is:

$$f = 0 \wedge s = 0 \wedge \tag{1}$$

$$(\exists f_0, s_0 : (f_0 = 1 \wedge s_0 = s \wedge f' = f_0 \wedge s' = 1) \vee \tag{2}$$

$$\exists f_0, s_0 : (f_0 = f \wedge s_0 = 1 \wedge f' = 1 \wedge s' = s_0)) \wedge \tag{3}$$

$$act' = 1 \tag{4}$$

Our relational semantics for an action is defined as $g \wedge B \wedge tr' = tr \frown [l]$. Equation 1 represents the guard g. Equations 2 and 3 represent the action's body B. Equation 2 refers to Line 8 of Example 1, Equation 3 to Line 11. Each represents a sequential composition: there must exist an intermediate state $\overline{v_0}$ (here f_0 and s_0) that can be reached from the first body predicate and from which the second body predicate can lead to its final state (f' and s'). Assignments update one variable with the value of an expression and leave the rest unchanged, e.g., the semantics of $f := 1$ at the end of Line 11 in Example 1 is $f' = 1 \wedge s' = s_0$. Equation 2 and 3 are connected via disjunction, which represents the non-deterministic choice between the sequential compositions. Finally, Equation 4 deals with the recording of the trace: each iteration of the do-od block may execute at most one action. Hence, our trace can be reduced to one variable act'. To make this formal semantics processable by constraint solvers, strings are encoded as integers, i.e., the label *'AlarmOn'* is represented by the constant 1. For the same reason, quantifiers are eliminated by substitutions (see [4] for details) resulting in the following constraints:

$$f = 0 \wedge s = 0 \wedge ((f' = 1 \wedge s' = 1) \vee (f' = 1 \wedge s' = 1)) \wedge act' = 1$$

This is how our tool generates the constraints. Further simplifications are possible, but not implemented: as both cases of the disjunction are equivalent, they may be reduced to one:

$$C^o = (f = 0 \wedge s = 0 \wedge f' = 1 \wedge s' = 1 \wedge act' = 1)$$

This expresses what was intended to be modelled: the action *'AlarmOn'* is executed if neither sound nor flash are activated yet and turns on both alarms. In disjunction with the semantics of the other actions, which are only indicated by ... in Example 1, C^o represents the transition relation of the action system.

Analogously, the simplified semantics of the mutation given in Example 1 is:

$$C^m = (f = 0 \wedge s = 0 \wedge ((f' = 0 \wedge s' = 1) \vee (f' = 1 \wedge s' = 1)) \wedge act' = 1)$$

2.2 Conformance Relation

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 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 our non-deterministic models an equivalence relation is not suitable as pointed out in [6]. An abstract non-deterministic model may do more than its concrete counterpart. Hence, useful conformance relations are relations relying on some ordering from abstract to more concrete models. One of these order relations is refinement, which uses implication to define conformance. A concrete implementation I refines an abstract model M, iff the implementation implies the model. The following definition of refinement relies on the Unifying Theories of Programming (UTP) of Hoare and He [25] giving M and I a predicative semantics.

Definition 1. ***(Refinement)*** *Let $v = \langle x, y, \dots \rangle$ be the set of variables denoting observations before execution and $v' = \langle x', y', \dots \rangle$ denoting the observations afterwards. Then*

$$M \sqsubseteq I \quad =_{df} \quad \forall v, v' : \; I(v, v') \;\Rightarrow\; M(v, v')$$

In [3] we 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 counterexamples to refinement. From Definition 1 follows that such counterexamples exist if and only if implication does not hold:

$$\exists v, v' \; : \; M^M(v, v') \wedge \neg M^O(v, v')$$

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

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 unsafe states. How we search for unsafe states in action systems is discussed in the next section.

3 Refinement Checking

In [5] we already gave an overview of our refinement checking approach. Fig. 2 depicts our process 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 combined via the non-deterministic choice operator. 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 3. ***(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 "reach" that returns the set of reachable states for a given trace in an action system. Then*

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

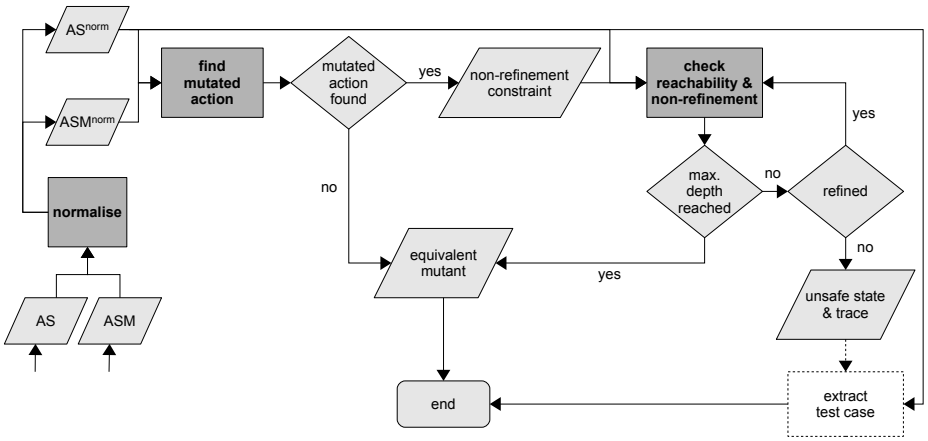

Fig. 2. Process for finding an unsafe state

This definition is different to Back's original refinement definition based on state traces [11]. 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:

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

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

Theorem 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:*

$$AS^O \not\sqsubseteq AS^M \textit{ iff } \bigvee_{i=1}^{n} \exists \overline{v}, \overline{v}', tr, tr' : (\overline{v} \in reach(AS^O, tr) \wedge A_i^M \wedge \neg A_1^O \wedge ... \wedge \neg A_m^O)$$

We use this property in our refinement checking process (Fig. 2), which is composed of several steps. At first, we *normalise* the original action system AS and the mutated action system ASM. We require action systems to be in a normal form corresponding to the disjunctive normal form (DNF) in predicate logic. This means that non-deterministic choice is always the outermost operator and not allowed in nested expressions. This is necessary for quantifier elimination, which is in turn required for the use with constraint solvers which do not support quantifiers in general. For a detailed description we refer to [4].

The next step *find mutated action* performs a syntactic check on AS^{norm} and ASM^{norm} to find out which action has been mutated. We then use Theorem 1 for which only one of the sub-constraints of the form $A_i^M \wedge \neg A_1^O \wedge \cdots \wedge \neg A_m^O$ can

be fulfilled: the one where A_i^M is the encoding of the just found mutated action. Hence, we are able to construct a *non-refinement constraint*, which is the sub-constraint containing the mutated action. It describes the set of unsafe states. Finally, the step *check reachability and non-refinement* performs a reachability analysis on the original action system and uses the non-refinement constraint to test each reached state whether it is an unsafe state. Note that the syntactic check fails to identify mutants that are syntactically inequivalent but behaviourally equivalent. These mutants could be skipped for reachability analysis and could be detected via a semantic analysis using Theorem 1. However, this semantic check has turned out the be more costly than the syntactic check followed by a reachability analysis [4].

Our process either results in the verdict *equivalent*, 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 derive a test case. As test case generation is out of the scope of this paper and remains future work, it is only indicated by a dotted box in Fig. 2. For a more details on our individual process steps and the used algorithms, we refer to [5, 4].

Example 3. We continue Example 1 to illustrate our refinement checking process. The action *'AlarmOn'* is already in normal form. Non-deterministic choice is the outermost operator and not nested in any other expressions. Our syntactic comparison reveals that the action *'AlarmOn'* has been mutated. The semantics of the original (C^o) and the mutant (C^m) have already been constructed in Example 2. By combining them, we get the non-refinement constraint:

$$C^m \wedge \neg C^o = (f = 0 \wedge s = 0 \wedge ((f' = 0 \wedge s' = 1) \vee (f' = 1 \wedge s' = 1)) \wedge act' = 1) \wedge$$

$$\neg (f = 0 \wedge s = 0 \wedge f' = 1 \wedge s' = 1 \wedge act' = 1)$$

Note that existential quantification of the state variables is implicitly performed by the constraint solver. The next step is the reachability analysis starting from the action system's initial state ($f = 0, s = 0$). Using this in the non-refinement constraint, we have:

$$(0 = 0 \wedge 0 = 0 \wedge ((f' = 0 \wedge s' = 1) \vee (f' = 1 \wedge s' = 1)) \wedge act' = 1) \wedge$$

$$\neg (0 = 0 \wedge 0 = 0 \wedge f' = 1 \wedge s' = 1 \wedge act' = 1)$$

By simplification, we get:

$$(((f' = 0 \wedge s' = 1) \vee (f' = 1 \wedge s' = 1)) \wedge act' = 1) \wedge \neg (f' = 1 \wedge s' = 1 \wedge act' = 1)$$

This constraint has one solution: $f' = 0, s' = 1, act' = 1$. It reveals wrong observations after the initial state. Originally, both f' and s' should be set to 1, but the mutant only sets s', but not f'. Hence, already the initial state ($f = 0, s = 0$) is an unsafe state and there is no need to continue reachability analysis in this simple example.

4 Efficiency

We presented techniques to improve the efficiency of our approach before [4]. Two of them are already incorporated in the description of our process in the previous section. The first one regards quantifier elimination. The second concerns the syntactic analysis to identify the mutated action. In the following, we present two further techniques to save computation time.

4.1 Pre-computation of Reachable States

So far, our refinement check between one original action system and a set of corresponding mutants has been implemented as described in Alg. 1. The input is one original action system (*as*) and a set of corresponding mutated action systems (*mutants*). Note that all action systems are supposed to be in normal form. The result is a map *unsafes* linking the mutants and their unsafe states. The algorithm iterates over the set of mutants (Line 4). For each mutant, it explores the state space of the original action system *as* (Line 6). The procedure *findNextState* implements a breadth-first search for successor states of state s, whereas it does not explore any state more than once (by maintaining a list of visited states). To ensure termination, it stops exploration at a user-specified depth limit. At each call, it returns the next reached state. This state is then tested whether it is an unsafe state (Line 8). If this is the case, the state space exploration is stopped and the next mutant is processed, where again state space exploration is performed. Note that we omitted the recording of the traces leading to the unsafe states for the sake of simplicity.

An advantage of Alg. 1 is that the state space is explored on demand, i.e., it is only explored until an unsafe state is found and not further. Given a small set of mutants, this algorithm is appropriate. When dealing with a large set of mutants, it is not that clever any more as the same state space is explored again and again. An alternative is to pre-compute all reachable states up to a given depth and then search for unsafe states in this set. Exploring the full state space to the maximum bound is not really an overhead as it has to be done for each equivalent mutant anyway. Given a large set of mutants, the probability that it contains at least one equivalent mutant is rather high.

Alg. 2 describes the refinement check with a pre-computed state space. It takes the same input as Alg. 1 and results in the same output. Again, all action systems are supposed to be in normal form. In contrast to Alg. 1, Alg. 2 only explores the state space once and then reuses the reached states during mutation analysis. The procedure *findAllStates* (Line 2) works analogously to the *findNextState* procedure of Alg. 1, but does not return one reachable state after the other. Instead, it returns the full set of reachable states at once. Afterwards, iteration over the mutants starts (Line 3), where each of the reached states (Line 4) is tested whether it is an unsafe state (Line 5). Once an unsafe state is found, we save the result (Line 6), stop searching for unsafe states (Line 7), and proceed with the next mutant without exploring the state space again.

Alg. 1 *chkRef*$(as, mutants) : unsafes$

```
 1: unsafes := [ ]
 2: visited := [ ]
 3: s := getInitState(as)
 4: for all asm ∈ mutants do
 5:    repeat
 6:       s := findNextState(as, s, visited)
 7:       visited := visited ∪ s
 8:    until unsafe(s, as, asm)
 9:    unsafes.add(asm, s)
10: end for
11: return unsafes
```

Alg. 2 *chkRef1*$(as, mutants) : unsafes$

```
 1: unsafes := [ ]
 2: states := findAllStates(as)
 3: for all asm ∈ mutants do
 4:    for all s ∈ states do
 5:       if unsafe(s, as, asm) then
 6:          unsafes.add(asm, s)
 7:          break
 8:       end if
 9:    end for
10: end for
11: return unsafes
```

4.2 Incremental Solving

Incremental solving is a technique to efficiently solve several constraints $c_1, ..., c_n$ that have large parts in common. The constraints are related to each other by the adding and/or the removal of small parts. Incremental solving exploits the findings made during solving the constraint c_i for solving the subsequent constraint c_{i+1} [34].

Our refinement checking process (Fig. 2) is well suited to exploit incremental solving. Alg. 2_incr is a more detailed version of Alg. 2 and gives additional details on the application of incremental solving. In Line 1, the original action system is translated. The resulting constraint system represents its transition relation (*trans_rel*). It is posted to the constraint/SMT solver's store (Line 2). The used interface for posting and retracting constraints acts as a stack, where constraints can be pushed or popped. Additionally, the interface provides a *solve* method. If the current store is satisfiable, it succeeds and a model may be retrieved. Otherwise, the constraints in the store are unsatisfiable and it returns false. The *solve* method is used in *findAllStates* (Line 3), where the transition relation of *as* is already in the solver's store. The procedure *findAllStates* starts at the initial state of *as* and recursively searches for all possible successor states. As the state space is now fully explored (up to a given depth limit), the transition relation is not needed any more and can be removed from the store (Line 4). In exchange, the negated transition relation is required for each refinement check with a mutant (cf. Theorem 1). It is added to the store in Line 5. The actual refinement check starts in Line 7. It iterates over the set of mutants. In Line 9, *findMutatedAction* syntactically compares the original and the mutated action system. Thereby, it identifies the mutated action a^m, which represents the second part of our non-refinement constraint (cf. Section 3). It is translated into constraints (Line 10) and added to the solver's store (Line 11), which now contains the complete non-refinement constraint for the current mutant. Lines 12 to 20 search for an unsafe state in the list of reachable states. Each state s is used as the pre-state v of the non-refinement constraint (Line 13). If it is satisfiable, we

Alg. 2_incr *chkRefIncremental*$(as, mutants)$: *unsafes*

```
 1: trans_rel := trans(as)
 2: solver.push(trans_rel)
 3: states := findAllStates(as, solver)
 4: solver.pop()
 5: solver.push(¬trans_rel)
 6: unsafes := [ ]
 7: for all asm ∈ mutants do
 8:    u := nil
 9:    a^m := findMutatedAction(as, asm)
10:    mut_act := trans(a^m)
11:    solver.push(mut_act)
12:    for all s ∈ states do
13:       solver.push(v = s) //s=unsafe?
14:       if solver.solve() then
15:          u := s
16:          solver.pop()
17:          break
18:       end if
19:       solver.pop()
20:    end for
21:    unsafes.add(asm, u)
22:    solver.pop()
23: end for
24: return unsafes
```

just found an unsafe state - a state from which the mutant behaves in a way that is not specified by the original (Lines 14 and 15). In this case we stop iterating over the states (Line 17). In any case, the constraint $v = s$ is removed from the store (Line 16 and Line 19 respectively). Line 21 inserts the mutant *asm* and the found unsafe state into the map *unsafes*. If no unsafe state could be found, *nil* is inserted and the mutant is considered to be equivalent up to the specified depth limit. In order to process the next mutant, the part of the non-refinement constraint that is specific to the current mutant has to be removed from the store (Line 22).

Alg. 2_incr shows that both the reachability analysis as well as the check for unsafe states are well suited to exploit incremental solving. During reachability, the transition relation is solved again and again - only the pre-states change (Line 3). While testing states whether they are unsafe, the non-refinement constraint has to be solved repeatedly - again with changing pre-states (Line13). Each non-refinement constraint contains the negated original (Line 5). Thus, when processing several mutants, there is a common part remaining in the store.

Incremental solving in Alg. 1 works analogously. Hence, we do not go into detail here. Note that we will use *Alg. 1_incr* to refer to the incremental version of Alg. 1 in the following.

5 Implementations

We implemented two versions of our refinement checking process (Fig. 2). The first version implements the repeated exploration of the state space, the second the pre-computation of the state space. Both incorporate incremental solving techniques. Hence, they implement Alg. 1_incr and Alg. 2_incr.

Our implementations are highly depending on the used solvers as they consume a large amount of our overall execution time. There's a strong competition within the constraint solving and SMT solving communities. Consider for exam-

ple the yearly SMT competition[1]. Therefore, the solvers are constantly enhanced to outperform others.

For this reason, we followed two parallel tracks in terms of solving techniques and also programming languages. The first implements the two algorithms in SICStus Prolog[2] (version 4.1.2) and uses the integrated constraint solver *clpfd* (Constraint Logic Programming over Finite Domains) [18]. The second track is implemented in the Scala programming language[3] and uses an SMT solver (Z3[4], version 4.0 for Mac OS X). While *clpfd* uses a specific input language, Z3 allows the use of SMT-LIB v2[5]. It is supported by most SMT solvers and hence facilitates the use of several different solvers. We also experimented with CVC3[6], MathSAT 5[7] and SMTInterpol[8], but Z3 was the most efficient for our problem. We restrict ourselves to linear integer arithmetic (QF_LIA logic in SMT-LIB).

In SICStus Prolog, the use of the constraint solver is very simple as its functionality is provided as a library. In order to use Z3 in Scala, we use JNA (Java Native Access)[9] to make the C API of Z3 accessible in Java. As Scala code is compiled to Java byte code, there is a strong interoperability with Java. Within Scala, Java libraries may be called directly and vice versa.

Incremental solving is directly supported by Z3. Its C API offers methods to push and pop clauses and to solve the clauses in the store. In Prolog, the *clpfd* library does not directly offer such an interface. Nevertheless, incremental solving can be implemented using constraint logic programming and backtracking.

In the following, we report on our experiences with our implementations.

6 Case Study: A Car Alarm System

To test our implementations, we used a simplified version of a car alarm system (CAS) by Ford. It already served as an industrial demonstrator in the MOGENTES project[10]. It can be considered as representative for embedded systems and has become a benchmark example within our research group.

Requirements. The following requirements 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 unauthorised person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

[1] `smtcomp.sourceforge.net`
[2] `http://www.sics.se/sicstus/`
[3] `http://www.scala-lang.org/`
[4] `http://research.microsoft.com/projects/z3/`
[5] `http://www.smtlib.org/`
[6] `http://www.cs.nyu.edu/acsys/cvc3/`
[7] `http://mathsat.fbk.eu/`
[8] `http://ultimate.informatik.uni-freiburg.de/smtinterpol/`
[9] `https://github.com/twall/jna`
[10] `https://www.mogentes.eu`

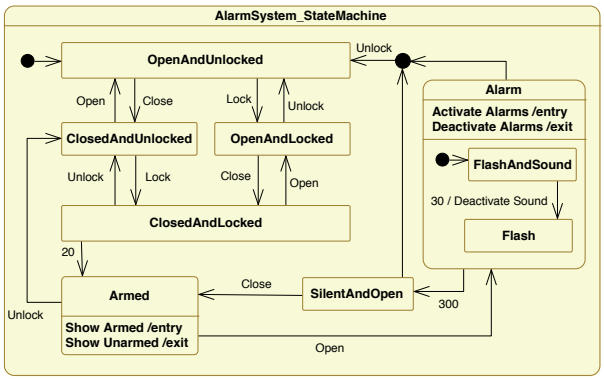

Fig. 3. UML state machine of the car alarm system

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.

Fig. 3 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 the *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. Here, the elapsing of time is modelled by explicit transitions using time triggers.

Mutations. We modelled the CAS described above as an action system and then manually created first order mutants of the model. We applied three mutation operators: (1) We set all possible guards to true (34 mutants). (2) We swapped equal and unequal operators (52 mutants). (3) We incremented all integer constants by 1, whereas we took the smallest possible value at the upper bound of a domain, in order to avoid domain violations (116 mutants). Additionally, we also included the original action system as an equivalent mutant. This gave us a total of 207 mutants.

Variations. As in our previous works [5, 4], we use four slightly different versions of our CAS model: (1) *CAS_1*: the CAS as introduced above with parameter values 20, 30, and 270 for waiting, (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.

Extended Model. Additionally, we built another extended version of the basic car alarm system (*CAS_1*). We parameterised the *Lock* and *Unlock* events with a PIN code. If the PIN code is correct when (un)locking the car, everything behaves as usual. If someone tries to (un)lock the car with an incorrect PIN code, the same mechanisms as when opening the car in the *Armed* state are triggered. This means that the system traverses to the *Alarm* state and the flash and sound alarms are turned on. Again, this state is left via timeouts or via unlocking the car with the correct PIN. For this *CAS_PIN* version, the timeouts are always 20, 30 and 270 seconds respectively. Nevertheless, there are also two variants: one with a Boolean PIN code and one with a PIN code consisting of three digits (0-999). Again, we applied our three mutation operators explained above, which gave us 245 mutants. We also included the original action system as an equivalent mutant resulting in a total of 246 mutants.

Technical Details. In the following, we present results on these six versions of the car alarm system obtained by (a) our Prolog implementation using a constraint solver and (b) our Scala implementation using the Z3 SMT solver. All of our experiments were conducted on a machine with a dual-core processor (2.8 GHz Intel Core i7) and 8 GB RAM with a 64-bit operating system (Mac OS X v10.7).

6.1 Results

Table 1 shows the runtimes for our CAS case study. All values are given in seconds and state the time needed for refinement checking of the original action system with all of its mutants. It gives numbers for the Prolog implementation using a constraint solver and the Scala implementation using an SMT solver (Section 5). The values in Table 1 are also illustrated in the diagram in Fig. 4.

The Prolog implementation yields the following results. Alg. 1_incr achieves runtimes from 2.82 to 8.55 seconds for the four CAS versions each with 207 mutants. For the two CAS versions with a PIN code (245 mutants in each case), it needs 3.52 and 5.93 seconds respectively. Hence, the runtimes are not constant with increasing domains of the parameters. Alg. 2_incr performs better. It is faster and the runtime is constant. Note that the most-constrained heuristic has been used for variable selection during constraint solving as our experiments in [4] indicate that it is a reasonable choice.

For the Prolog implementation, we reported earlier results for CAS_1 to CAS_1000 in [4]. At that time, our implementation used Alg. 1, i.e., it did not exploit incremental solving and did not pre-compute and reuse the state space. The computation times for refinement checking of 207 mutants were about 20 seconds for each CAS version. Hence, incremental solving led to faster runtimes, but reduced scalability. By Alg. 2_incr, constant runtimes could re-established.

Table 1. Runtimes (sec) for the CAS obtained by our two implementations

		CAS				CAS_PIN	
		1	10	100	1000	Bool	0-999
Prolog &	Alg. 1_incr	2.82	2.84	3.38	8.55	3.52	5.93
Constraint Solving	Alg. 2_incr	2.63	2.62	2.66	2.69	3.28	3.27
Scala &	Alg. 1_incr	18.13	17.60	17.69	17.72	21.05	22.23
SMT Solving	Alg. 2_incr	4.18	4.15	4.13	4.11	5.28	5.33

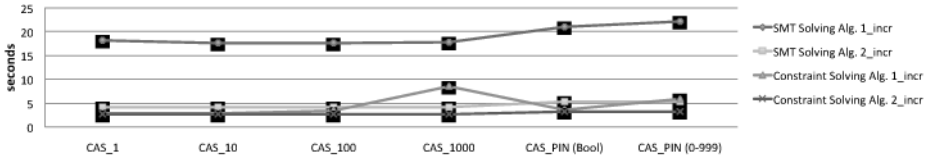

Fig. 4. Comparison of the runtimes for the car alarm system case study

Thereby, we achieve a significant performance gain by our latest improvements (see Section 4). The execution time could be improved by 85% compared to our latest results published in [4].

Using the Scala implementation of Alg. 1_incr and the Z3 SMT solver, it takes approximately 18 seconds for *CAS_1* to *CAS_1000*. The two CAS versions extended with a PIN code require about 22 seconds each. For Alg. 2_incr, we have runtimes of 4 seconds for *CAS_1* to *CAS_1000* and 5 seconds for the two *CAS_PIN* versions. In contrast to the implementation based on constraint solving, the SMT-solver-based implementation always shows constant runtime. Again, Alg. 2_incr performs better than Alg. 1_incr.

Using Alg. 2_incr, both implementations achieve almost equally fast runtimes. For Alg. 1_incr, the SMT-solver-based implementation is slower, but still reasonable. This cannot solely be explained by the use of different solving techniques (constraint vs. SMT solving) as we discuss in the following.

6.2 Discussion

We implemented two refinement checking tools for action systems and compared their runtimes above. Both tools result in the same counterexamples and unsafe states, but are implemented differently. They use different techniques for problem solving. One relies on constraint solving, the other one on SMT solving. Nevertheless, the comparison cannot only be reduced to this. There are other factors influencing the performance of the implementations. They are written in different programming languages. On the one hand, the logic programming language Prolog was used. Its main advantage for our application is the native support for constraint solving. Due to its backtracking facilities, also incremental solving could be implemented quite efficiently. On the other hand, Scala is used. It is based on the Java Virtual Machine (JVM) and is a multi-paradigm language combining object-oriented and functional programming. In contrast to the Prolog implementation, the program written in Scala depends on an external SMT

solver that is only accessible via non-Java APIs (C, Python, ...). This introduces some overhead compared to the native constraint solving support in Prolog. Additionally, the implementations were written by different programmers. The individual styles of programming may also have influenced the comparability of the results.

We are aware that the results from our case study may not generalise. We reported on our experience with this kind of models, which we think are representative for most typical embedded systems. Nevertheless, the scalability of our approach on the CAS does not necessarily have to be representative for other types of models.

7 Related Work

To our knowledge, our test case generation approach is the first that deals with non-deterministic systems, uses mutations, and is based on constraint solving techniques. Nevertheless, there exist various works overlapping in one or several aspects. There are constraint-based test case generation approaches on the source code level, where no non-determinism has to be considered. A mutation-based approach is [35] for example. Java-like programs are mutated and transformed into constraints via SSA form to generate distinguishing test cases. Gotlieb et al. do not use mutations, but structural criteria for test data generation via SSA form. In [22], they work with constraint solving, in [23] with CLP.

Regarding black-box techniques, one of the first models to be mutated were predicate-calculus specifications [17] and formal Z specifications [31]. 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 [8]. Most test case generation approaches using model checkers deal with deterministic systems. Nevertheless, there also exist works considering non-determinism and the involved difficulties. [28] suggests to synchronise non-deterministic choices in the original and the mutated model via common variables to avoid false positive counterexamples. [15] proposes two approaches that cope with non-determinism: modular model checking of a composition of the mutant and the specification, and incremental test generation via observers and traditional model checking. [27] also considers non-determinism. It uses the model checker/refinement checker FDR (Failures-Divergence Refinement) for the CSP process algebra [29] to generate test cases. However, this approach is not mutation-based.

Other model-based mutation testing techniques considering non-determinism include two ioco (input-output conformance [32]) checkers for LOTOS specifications [7] and (qualitative) action systems [16]. Both are not symbolic, but rely on explicit state space enumeration.

In our symbolic approach we use constraint/SMT solvers that support incremental solving. Incremental solving has already been applied to many problem domains. One very prominent application is bounded model checking [34, 30].

8 Conclusion

We enhanced our refinement checking approach for non-deterministic action systems in two ways. Firstly, we presented a more efficient way of processing a large number of mutants. Secondly, we exploited incremental solving techniques. We implemented both improvements in two implementation tracks: the first one uses Prolog and constraint solving, the second Scala and the SMT solver Z3. First case studies with a car alarm system showed that our two improvements significantly reduced runtime. Previous results [4] could be improved by up to 85%. Our case study also indicates that both SMT and constraint solvers are able to cope with refinement checking problems.

The ultimate goal of our work is to enable test case generation that targets specific faults. Thereby, a mutation adequacy score that is as high as possible shall be achieved. This paper dealt with the necessary, underlying conformance check. Future work will attach to this and generate the desired test cases.

Of course, we are aware that our results may not generalise. To give further evidence for the effectiveness and scalability of our approach, we already work on further case studies. Regarding effectiveness, one very important aspect of our model-based mutation testing approach are the used fault models. So far, we only applied three manual mutation operators for action systems. We aim for integrating our refinement checking implementations into an already existing framework. It uses UML models, which are mutated and then translated into action systems. In [1, 2], this framework has already been used successfully with an explicit conformance checking tool. Nevertheless, scalability was an issue and motivated this work. Regarding scalability of our new approach, also counterexample-guided abstraction refinement techniques similar as in *SLAM* [12] or *BLAST* [13] may be an interesting topic for future work.

Acknowledgments. 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. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Efficient mutation killers in action. In: IEEE 4th Int. Conf. on Software Testing, Verification and Validation, ICST 2011. pp. 120–129. IEEE Computer Society (2011)
2. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: UML in action: A two-layered interpretation for testing. In: UML&FM 2010: Third IEEE Int. workshop UML and Formal Methods. ACM Software Engineering Notes (SEN) (2011)
3. Aichernig, B.K., He, J.: Mutation testing in UTP. Formal Aspects of Computing 21(1-2), 33–64 (2009)
4. Aichernig, B.K., Jöbstl, E.: Efficient refinement checking for model-based mutation testing. In: 12th Int. Conf. on Quality Software (QSIC 2012). pp. 21–30. IEEE Computer Society (2012)

5. Aichernig, B.K., Jöbstl, E.: Towards symbolic model-based mutation testing: Combining reachability and refinement checking. In: 7th Workshop on Model-Based Testing (MBT 2012). EPTCS, vol. 80, pp. 88–102 (2012)
6. Aichernig, B.K., Jöbstl, E.: Towards symbolic model-based mutation testing: Pitfalls in expressing semantics as constraints. In: Workshops Proc. of the 5th Int. Conf. on Software Testing, Verification and Validation (ICST 2012). pp. 752–757. IEEE Computer Society (2012)
7. Aichernig, B.K., Peischl, B., Weiglhofer, M., Wotawa, F.: 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). pp. 215–224. IEEE Computer Society (2007)
8. Ammann, P., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: 2nd IEEE Int. Conf. on Formal Engineering Methods (ICFEM 1998). pp. 46–54. IEEE Computer Society (1998)
9. Back, R.J., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: 2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing. pp. 131–142. ACM (1983)
10. Back, R.J., Kurki-Suonio, R.: Distributed cooperation with action systems. ACM Transactions on Programming Languages and Systems 10(4), 513–554 (1988)
11. Back, R.J., Sere, K.: Stepwise refinement of action systems. Structured Programming 12, 17–30 (1991)
12. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN 2001 Workshop on Model Checking of Software. LNCS, vol. 2057, pp. 103–122. Springer (2001)
13. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST. Int. Journal on Software Tools and Technology Transfer 9, 505–525 (2007)
14. Bonsangue, M.M., Kok, J.N., Sere, K.: An approach to object-orientation in action systems. In: Mathematics of Program Construction (MPC'98). LNCS, vol. 1422, pp. 68–95. Springer (1998)
15. Boroday, S., Petrenko, A., Groz, R.: Can a model checker generate tests for non-deterministic systems? Electronic Notes in Theoretical Computer Science 190(2), 3–19 (2007)
16. Brandl, H., Weiglhofer, M., Aichernig, B.K.: Automated conformance verification of hybrid systems. In: 10th Int. Conf. on Quality Software (QSIC 2010). pp. 3–12. IEEE Computer Society (2010)
17. Budd, T.A., Gopal, A.S.: Program testing by specification mutation. Computer languages 10(1), 63–73 (1985)
18. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: 9th Int. Symp. on Programming Languages: Implementations, Logics, and Programs. pp. 191–206. PLILP '97, Springer (1997)
19. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: Help for the practicing programmer. IEEE Computer 11(4), 34–41 (April 1978)
20. Dijkstra, E., Scholten, C.: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science, Springer-Verlag (1990)
21. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. 19(3), 215–261 (Sep 2009)
22. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: Int. Symp. on Software Testing and Analysis (ISSTA). pp. 53–62 (1998)

23. Gotlieb, A., Botella, B., Rueher, M.: A CLP framework for computing structural test data. In: 1st Int. Conf. on Computational Logic (CL 2000). LNCS, vol. 1861, pp. 399–413. Springer (2000)
24. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering 3(4), 279–290 (July 1977)
25. Hoare, C., He, J.: Unifying Theories of Programming. Prentice-Hall Int. (1998)
26. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
27. Nogueira, S., Sampaio, A., Mota, A.: Guided test generation from csp models. In: 5th Int. Colloquium on Theoretical Aspects of Computing (ICTAC 2008). LNCS, vol. 5160, pp. 258–273. Springer (2008)
28. Okun, V., Black, P.E., Yesha, Y.: Testing with model checker: Insuring fault visibility. In: Mastorakis, N.E., Ekel, P. (eds.) 2002 WSEAS Int. Conf. on System Science, Applied Mathematics & Computer Science, and Power Engineering Systems. pp. 1351–1356 (2003)
29. Roscoe, A. W.: Model-checking CSP, chap. 21. Prentice-Hall (1994)
30. Shtrichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: 11th IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods. pp. 58–70. CHARME '01, Springer (2001)
31. Stocks, P.A.: Applying formal methods to software testing. Ph.D. thesis, Department of computer science, University of Queensland (1993)
32. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools 17(3), 103–120 (1996)
33. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Software Testing, Verification and Reliability (2011)
34. Whittemore, J., Kim, J., Sakallah, K.: SATIRE: A new incremental satisfiability engine. In: Proc. of the 38th annual Design Automation Conf. pp. 542–545. DAC '01, ACM (2001)
35. Wotawa, F., Nica, M., Aichernig, B.K.: Generating distinguishing tests using the Minion constraint solver. In: Workshops Proc. of the 3rd Int. Conf. on Software Testing, Verification and Validation (ICST 2010). pp. 325–330. IEEE Computer Society (2010)