Efficient Refinement Checking for Model-Based Mutation Testing

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

Abstract—In model-based mutation testing, a test model is mutated for test case generation. The resulting test cases are able to detect whether the faults in the mutated models have been implemented in the system under test. For this purpose, a conformance check between the original and the mutated model is required. We have developed an approach for conformance checking of action systems, which are well-suited to specify reactive and non-deterministic systems. We rely on constraint solving techniques. Both, the conformance relation and the transition relation are encoded as constraint satisfaction problems. Earlier results showed the potential of our constraint-based approach to outperform explicit conformance checking techniques, which often face state space explosion. In this work, we go one step further and show optimisations that really boost our performance. In our experiments, we could reduce our runtimes by 80%.

Keywords—action systems; conformance; refinement; model-based testing; mutation testing; constraint solving

I. INTRODUCTION

Mutation testing is a fault-based software testing technique that receives growing interest [1]. Yet, it is still widely perceived as being costly and impractical. This remains a barrier to wider uptake within industry. In this paper, we report how these costs can be reduced for a particularly hard problem in mutation testing: the generation of test cases from mutated, non-deterministic models.

Mutation testing is a technique for assessing and improving a test suite [2], [3]. A number of faulty versions of a program under test are produced by injecting bugs into its source code. These faulty programs are called mutants. A tester analyses if his test suite can kill all mutants. We say that a test kills a mutant if it is able to distinguish it from the original. The tester improves his test suite until all faulty mutants get killed.

Unfortunately, the method is not that straightforward, because not all mutants are faulty, i.e. not all injected faults cause observable failures. For example, injected faults in dead code have no effect. Such mutants cannot be killed and are therefore called equivalent mutants. These mutants need to be singled out by other means than testing. Traditionally, this has been done by manual inspection, because program equivalence is undecidable in general. However, as we demonstrate in this paper, modern program verification techniques can efficiently deal with equivalent mutants.

In this work, we concentrate on model-based mutation testing. It combines ideas from mutation testing and model-based testing. Model-based testing is a black-box testing technique that avoids the labour of manually writing hundreds of test cases, but instead advocates the capturing of the expected behaviour in a model of the system under test (SUT). The test cases are automatically generated from this model [4]. The technique is receiving growing interest in the embedded-systems domain, where models are the rule rather than the exception [5].

In model-based mutation testing, we view the SUT as a black box. Hence, we have no access to the source code and consequently, cannot mutate it. Therefore, we mutate a model of the SUT. This original model is assumed to be correct with respect to some properties derived from the requirements. This can be assured, e.g. via model checking. Then, given the original model and a set of mutated models, we automatically generate test cases that kill the model mutants. Equivalent mutants are singled out automatically. Hence, in contrast to program mutation, where we analyse a given set of test cases, here we generate a test suite that will kill all (non-equivalent) mutants. This is non-trivial, since it involves an equivalence check between original and mutated models. Since, equivalence is undecidable in general, we restrict ourselves to bounded domains. How such an efficient checker can be implemented with a constraint solver is the topic and main contribution of this paper.

The situation is even more interesting when we consider non-deterministic models. In a non-deterministic model, a given (sequence of) input stimuli may cause several possible output observations. This is due to abstraction in the test models. When comparing two non-deterministic models, an original and a mutant, equivalence is not sufficient. We need an order or preorder relation. Refinement is such an order relation [6]. In this paper, we show how a refinement checker can effectively analyse a large number of mutated models.

Compared to our previous work [7], [8], we have reduced the test case generation time by 80%. The specific contributions of this work are the optimisation techniques, their implementation via a constraint solver, and the detailed experimental results.

This paper is organised as follows: Section II introduces preliminaries, i.e., our modelling language and our notion of refinement. Section III explains the principles of refinement checking and Section IV focuses on our techniques for increasing its efficiency. Section V presents our experimental results. Finally, in Section VII we draw our conclusions.
II. Preliminaries

A. 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 [12], 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.

Syntax: There exist various versions of Back’s original action system notation [9]. The syntax we use is defined in Figure 1. 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 \( A \), 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 \( \vec{v} \) and the definition of the initial state as a vector of constants \( \vec{c} \). An action \( A \) is a labelled guarded command with label \( L \), guard \( g \) and body \( B \). Actions may have a list of parameters \( X \). 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 \( \mid \). 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.

Semantics: The semantics of programs is often encoded via static single assignment (SSA) form [13]. For action systems, the formal semantics is typically defined in terms of weakest preconditions. However, for our constraint-based approach, we have not stuck to any of these two. We found a relational predicative semantics being more suitable. Hence, we follow the style of He and Hoare’s Unifying Theories of Programming [14]. In [8] we gave reasons for our choice. Figure 2 presents our formal semantics of the actions of our modelling language. The state-changes of actions are defined via predicates relating the pre-state of variables \( \vec{v} \) and their post-state \( \vec{v}' \). 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 \( X \), 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 \( \vec{v}'' \) 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.

B. 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 no suitable conformance relation as pointed out in [8]. 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 this 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 [14] giving \( M \) and \( I \) a predicative semantics.

Definition 1. (Refinement)

\[
M \sqsubseteq I = \forall x, x', y, y', \ldots \in \alpha : I \Rightarrow M
\]

for all \( M, I \) with alphabet \( \alpha \).

The alphabet \( \alpha \) is the set of variables denoting observations. Unprimed variables represent variables before execution, primed variables denote observations afterwards.

In [15] 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 counter-examples to refinement. From Definition 1 follows that such counter-examples exist if and only if implication does not hold:

\[
\exists x, x', y, y', \ldots \in \alpha : M^M \land \neg M^O
\]

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

\[
\text{valid} = \{ s \mid \exists s' : M' (s,s') \land \neg M (s,s') \}
\]

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

III. REFINEMENT CHECKING

In [7] we already gave an overview of our refinement checking approach. Figure 3 depicts our process to find an unsafe state. The inputs are the original action system model \( M \) and a mutated version \( M' \). Each action system consists of a set of actions \( A \) and \( A' \) respectively, which are combined via the non-deterministic choice operator. The observations in our action language are the event traces and the system states before \( (\bar{v}, t) \) and after one execution \( (\bar{v}', t') \) of the do-od block. Then, a mutated action system \( M' \) refines its original version \( M \) 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 \( M \) and \( M' \) be two action systems with corresponding do-od blocks \( P \) and \( P' \). Furthermore, we assume a function “reach” that returns the set of reachable states for a given trace in an action system. Then

\[
\text{reach} : \text{init} \rightarrow \text{state}
\]

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:

\[
\exists \bar{v}, \bar{v}', t, t' : (\bar{v} \in \text{reach}(A \land t) \land \neg (A_i \lor \cdots \lor A_n) )
\]

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 \( M' \) does not refine its original action system \( M \), iff any action \( A_i \) of the mutant shows trace or state-behaviour that is not possible in the original action system:

\[
\exists \bar{v}, \bar{v}', t, t' : (\bar{v} \in \text{reach}(A \land t) \land \neg A_i \lor \cdots \lor \neg A_n )
\]

We use this property in our refinement checking process, which is composed of several steps. The process step find mutated action is a preprocessing activity to check for refinement regardless of reachability. It uses Theorem 1 for which it is sufficient to satisfy one of the sub-constraints of the form \( A_i \lor \cdots \lor A_n \) in order to find non-conformance. Hence, it is possible to perform a non-refinement check action by action of the mutant. Thereby, we pass smaller constraint systems to the solver. If there does not exist an unsafe state at this point, we cannot find any mutated action that yields non-conformance and 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 is the sub-constraint that could be satisfied. It describes the set of 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. Note that the step normalise in the beginning of the process is new and will be explained below. We implemented our technique in SICStus Prolog\(^1\) (version 4.1.2). SICStus comes with an integrated constraint solver clpfd (Constraint Logic Programming over Finite Domains) [16], 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, which we have not implemented yet. Test case generation is indicated by the dotted parts at the right bottom of Figure 3. For a more detailed description of our individual process steps and the used algorithms, we refer to [7]. In the following, we concentrate on improvements of our preliminary approach in order to increase its efficiency.

**IV. EFFICIENT REFINEMENT CHECKING**

**A. Quantifier Elimination and Normal Form**

A standard approach for turning a program into a constraint satisfaction problem is to convert it into SSA form at first and then replace sequential composition by conjunction. As already pointed out in [8], this is not possible in our refinement check due to the required negation. Hence, we have to use the general definition of sequential composition, which is also used in He and Hoare’s Unifying Theories of Programming [14]. Thereby, we gain a relational predicative semantics (see Figure 2), where sequential composition is expressed as follows: there must exist an intermediate state \(v_0\) that can be reached from the first body predicate and from which the second body predicate can lead to its final state. If sequential composition is used in the original action system, it has to be negated for our non-refinement check (see Section II-B). Hence, we get the following constraints:

\[
\neg (\exists v_0 : (B(v, v_0) \land B(v_0, v')))
\]

By resolving negation, we get

\[
\forall v_0 : (\neg (B(v, v_0) \land B(v_0, v')))
\]

This constraint system uses universal quantification, which is not supported by common constraint solvers. Fortunately, we are able to resolve this problem by application of the so-called one-point rule:

\[
(\exists x : x = e \land P(x)) \iff P(e)
\]

This means that if the variable is fixed to one value, it is possible to substitute the value for the variable and eliminate existential quantification. Note that our semantics incorporate identity assignments

\[
x := e =_{df} x' = e \land y' = y \land \ldots \land z' = z
\]

(cf. Figure 2). In this way, no variable assignment is lost by this substitution, which also constitutes an optimisation as the constraints passed to the constraint solver are reduced in terms of the number of used variables.

However, the application of the one-point rule is only possible if the left-hand side of sequential composition is deterministic, i.e., binds a variable to one value. This is the case for assignments. Nevertheless, constructs like

\[
\exists out_0 : ((out_0 = 1 \lor out_0 = 2) \land out' = out_0 + 1)
\]

are possible. Here, the left-hand side of sequential composition is not deterministic and we cannot substitute since we do not know which value will be assigned to \(out\). We can avoid such problems by introducing a normal form which requires that non-deterministic choice is always the

\[^1\text{http://www.sics.se/sicstus/}\]
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. In predicate logic, this required normal form corresponds to the disjunctive normal form (DNF). Hence, each action system can be automatically normalised.

We implemented the normalisation and the application of the one-point rule in Prolog: as depicted in Figure 3, we first normalise the action systems. Non-deterministic choice is then always the outermost operator. Hence, all branching happens at the beginning of each iteration through the do-od block. The application of the one-point rule is implemented via symbolic execution during the translation of the do-od block. Note that our symbolic execution is simpler than in the general case as no branching occurs in between. In each guard, we replace all references to variables by their current symbolic values. Consecutive guards are combined via conjunction leading to our path condition. At each assignment, we update the symbolic value for the assigned variable. At the end of each path, the final symbolic value for each variable \( v_i \) is added to the path condition \( pc \). Having a lookup function \( symbVal \) that takes a variable and returns its symbolic value, we have:

\[
pc \land v_1 = symbVal(v_1) \land \ldots \land v_n = symbVal(v_n)
\]

As an example, consider the following code snippet of an action \( set\_a\_b \) that sets two Boolean variables \( a \) and \( b \) in an arbitrary order. Note that \( \#= \) denotes our equality operator and \( := \) the assignment operator.

1. \( set\_a\_b::(true) \Rightarrow (\)
2. \( ((a \#= 0 \Rightarrow a := 1)) ; (b \#= 0 \Rightarrow b := 1)) \)
3. \( (a \#= 0 \Rightarrow a := 1)) ; (b \#= 0 \Rightarrow b := 1)) \)

The arbitrary order is modelled by the sequential composition (\()\) of two non-deterministic choices (\[]\). Line 2 non-deterministically either sets \( a \) or \( b \). Subsequently, the other variable (the one which is not yet 1) is set. After normalisation, we have:

1. \( set\_a\_b::(true) \Rightarrow (\)
2. \( (a \#= 0 \Rightarrow a := 1)) ; (a \#= 0 \Rightarrow a := 1) \)
3. \( (a \#= 0 \Rightarrow a := 1)) ; (b \#= 0 \Rightarrow b := 1) \)
4. \( (b \#= 0 \Rightarrow b := 1)) ; (a \#= 0 \Rightarrow a := 1) \)
5. \( (b \#= 0 \Rightarrow b := 1)) ; (b \#= 0 \Rightarrow b := 1) \)

Given unprimed variables \( a \) and \( b \) as initial values and primed variables \( a' \) and \( b' \) as final values, our translation of this action yields the following constraints:

1. \( a = 0 \land 1 = 0 \land a' = 1 \land b' = b \)
2. \( \lor a = 0 \land b = 0 \land a' = 1 \land b' = 1 \)
3. \( \lor b = 0 \land a = 0 \land a' = 1 \land b' = 1 \)
4. \( \lor b = 0 \land 1 = 0 \land a' = a \land b' = 1 \)

Note that the encoding of the action to build the trace has been omitted in this example. Obviously, the first and last constraints are unsatisfiable (1 = 0). They represent infeasible paths. We report results regarding the runtime of the just described implementation later on (Section V-C).

B. Variable and Value Selection Heuristics

So far, our implementation always used the default settings of SICStus Prolog’s integrated constraint solver clpfd (Constraint Logic Programming over Finite Domains) [16]. One obvious action that should be tried out to improve performance is to adjust these settings. We modified the search strategy of the constraint solver by trying different combinations of variable and value selection strategies. By default, variables are selected from left to right (leftmost), Other variable selection strategies include the first-fail principle (ff), which selects the variable with the smallest domain, and the most-constrained heuristic (ffc), which selects the variable that has the smallest domain and the most constraints suspended on it. For value selection, the default is to try values in ascending order (up). The other alternative is to use descending order (down). By trying all combinations of these variable and value selection strategies, we get six possible settings. We report results later in Section V-D.

C. Mutation Detection Strategies

It is possible to reduce the size of the constraint systems that have to be processed by the constraint solver. As already explained above, the non-refinement condition presented in Theorem 1 is a disjunction of constraints of which each deals with one action \( A_i^M \) of the mutated action system \( AS_i^M \). Hence, it is sufficient to satisfy one of these sub-constraints in order to prove non-conformance. We have to find the mutated action \( A_i^M \) in order to construct the non-refinement constraint representing the set of unsafe states. For actions that have not been mutated, this set would be empty. So far, mutation detection has been realised by passing our non-refinement constraints to the constraint solver one after the other for each action of the mutated action system. This has the advantage that only “real” semantical mutations are detected, but is rather demanding in terms of runtime. A simpler and faster way to identify which action has been altered is to perform mutation detection on a syntactic level, i.e. by comparing the source code of the actions.

For this syntactic analysis, we have to consider the definitions of the actions. Additionally, their calls in the do-od block are important, where parameters could be manipulated. For example, parameters could be replaced by constant values or other variables. Note that our syntactic comparison is not sensitive to the pure renaming of parameters and local variables, which are represented by Prolog variables. This is implemented easily by using SICStus Prolog’s library of term utilities. The predicate variant/3 checks whether two terms are identical modulo renaming of variables.

Our syntactic check requires some pre-conditions to be fulfilled. First of all, we do not support overloading of actions. This is, each action in an action system is uniquely identified by its name. There must not exist two actions having the same name but a different number of parameters. Furthermore, we suppose that no action call is completely
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.

B. Experimental Setup

For an experimental evaluation of our improved implementation, we have basically repeated our experiments from [7]. We modelled the CAS described above as an action system and then manually created first order mutants (one mutation per mutant) 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). Furthermore, we also included the original action system as an equivalent mutant. This gave us a total of 207 mutants. Note that here the setup of our experiments slightly differs from [7], where we had to exclude 12 mutants since the constraints given to the solver could not be processed within a reasonable amount of time. This time, this is not necessary. All 207 mutants are subject to our experiments.

As already described in [7], there exist four slightly different versions of our CAS model: (1) CAS_1: the CAS as presented in Section V-A 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. Our experiments were conducted for all of these four CAS versions on a machine with a dual-core processor (2.8 GHz) and 8 GB RAM with a 64-bit operating system.

In the following, we present the results from our improved refinement checker, compare them with the results obtained with the preliminary version presented in [7], and with results obtained with Ulysses [17], [18]. Ulysses is a mutation-based test case generator and works with action systems, too. Nevertheless, this comparison is not totally fair as already explained in [7]: (1) Ulysses generates adaptive test cases, not only a trace leading to an unsafe state as our tool does. (2) Ulysses uses a different conformance relation named ioco (input-output conformance [19]). Therefore, we ran Ulysses in two settings. First, on the CAS with distinguished

deleted from the do-od block and that no call is added by mutation operators. Finally, we do not allow the mutation of data types. If some of these pre-conditions are not fulfilled, it is possible that we miss a mutation. Our implementation checks only the last assumption: each type defined in both, the original and the mutated action system must have the same definition. The other assumptions are not checked automatically. Hence, our mechanism for finding the mutated action is implemented in a conservative way: If we cannot find any mutated action syntactically (which might happen if our pre-conditions are not fulfilled), we perform our semantic mutation detection strategy. We report our experimental results with syntactic mutation detection in Section V-E.

V. EXPERIMENTAL RESULTS

A. Car Alarm System

For our experiments, we used a simplified version of a car alarm system (CAS). The following requirements served as the basis for our model:

R1 Armimg. 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.

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.

Figure 4 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.
input and output actions. 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 have this distinction between input and output actions either. However, the conformance relations are still not identical.

C. Quantifier Elimination and Normal Form

In [7], we presented preliminary results of our refinement checker. At that time, we could not cope with all 207 mutants. For 12 mutants, the constraints given to the solver could not be processed within a reasonable amount of time. The remaining 195 mutants could be handled in about 100 seconds. This is an average of 0.5 seconds per mutant. The maximum runtime for one mutant was around 3 seconds.

Table I lists the results with our improved refinement checker, which normalises the action systems and applies the one-point rule during their translation resulting in constraints using less variables. As already mentioned in Section V-B, we conducted our experiments for four versions of our car alarm system: CAS_1, CAS_10, CAS_100, and CAS_1000. For each, we give the time needed to process all 207 mutants (Σ), the average time needed for one mutant (ϕ), and the maximum amount of time needed for one mutant (max).

For our refinement checker, we divide the execution time into two parts: (1) the time to find the mutated action, i.e. for checking whether there possibly exists an unsafe state and which action has been mutated (column 1: find mutated action), and (2) the time needed for the combined reachability and non-refinement check (column 2: reach & non-refine). The sum thereof results in the overall execution time for refinement checking (column total). Table I also lists the runtimes for the explicit loco checker Ulysses. As already explained in Section V-B, we ran Ulysses in two settings: (1) with distinguished input and output actions (column in/out), and (2) with all actions being classified as outputs (column out).

With our improved refinement checker it is possible to deal with all 207 mutants, which was not possible before (see [7]). Still, the overall execution time could be improved from 100 seconds to 41 seconds for the basic version of the CAS (CAS_1). This means that our improved implementation is more than twice as fast. It is also faster than Ulysses, which needs 103 seconds with distinguished inputs and outputs and 68 seconds if all actions are classified as outputs. Nevertheless, this version of our refinement checker does not scale any more. For CAS_10 it needs already 179 seconds, then half an hour for CAS_100, and finally approximately 4 hours for CAS_1000. Note that Table I does not contain minimum values as they are 0 ms for our refinement checker, i.e. not measurable in practice. For Ulysses, they are also below 0.5 seconds.

When comparing our refinement checker improved by the one-point-rule with the one used in [7], we can make another observation. The older version was rather fast in finding the mutated action - about 15 seconds for the 195 mutants for each CAS version. This was not much compared to the time needed for the combined reachability and non-refinement check, which took about 90 seconds for the 195 mutants. This has changed for our new implementation. Now, finding the mutated action takes more than half of the time for the total refinement checking process and this already for the small CAS_1. For the largest CAS (CAS_1000), it takes even 4.2 h for the 207 mutants, while the combined reachability and non-refinement check stays rather constant (18 - 23 seconds per CAS version for all mutants).

D. Variable and Value Selection Heuristics

As already explained in Section IV-B, we experimented with the constraint solver’s search strategy by trying different combinations of variable and value selection strategies. Variable selection strategies include: (1) leftmost, which selects variables from left to right, (2) ff - the first-fail principle, and (3) ffc - the most-constrained heuristic. For value selection, up or down may be chosen, i.e. values are either selected in ascending or in descending order.

By trying all combinations of these variable and value selection strategies, we get six possible settings. The runtimes of our refinement checker using the default setting (leftmost-up) were already reported in Table I. The execution times in seconds for the others are listed in Table II. Again, we partition the total runtime into the time needed for finding the mutated action “I” and the time for the combined reachability and non-refinement check “2”. Note that the default setting leftmost-up is the worst setting for our example. It takes up to 3.4 hours for CAS_1000 (see Table I). As can be seen from Table II, also leftmost-down does not scale for larger parameter domains. In general, the first-fail principle (ff) as well as the most-constrained heuristic (ffc) show good results regardless of the value selection strategy and the size of the variable domains. This makes sense as these heuristics are more sophisticated than the static selection of the leftmost variable. They adapt to the current state of the search and select the next variable dynamically. Nevertheless, the combination leftmost-down may also accomplish good results. For example, it achieves the shortest runtimes (27 and 31 seconds respectively) for CAS_1 and CAS_10. For CAS_100 and CAS_1000, ffc-up, i.e. the combination of the most-constrained heuristic and the ascending order for value selection, is the fastest combination. So there is no setting that performs best for all CAS versions. Nevertheless, using the first-fail principle or the most-constrained heuristic seems to be the best choice in general. They scale for all four CAS versions (although CAS_100 seems to be a small outlier).

Note that we also tried the different variable and value selection strategies for our “old” refinement checker presented in [7]. Our experiments showed that it was still not possible to process all 207 mutated action systems within
Table I
EXECUTION TIMES FOR OUR REFINEMENT CHECKING TOOL IMPROVED BY APPLYING THE ONE-POINT RULE AND THE ioco CHECKER ULYSSES APPLIED ON FOUR VERSIONS OF THE CAR ALARM SYSTEM. ALL VALUES ARE GIVEN IN SECONDS UNLESS OTHERWISE NOTED.

<table>
<thead>
<tr>
<th>CAS version</th>
<th>leftmost-down 1</th>
<th>leftmost-down 2</th>
<th>total 1 2</th>
<th>total 1 2</th>
<th>total 1 2</th>
<th>total 1 2</th>
<th>total 1 2</th>
<th>total 1 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS_1</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>31</td>
<td>18</td>
<td>41</td>
<td>103</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.09</td>
<td>0.13</td>
<td>0.15</td>
<td>0.09</td>
<td>0.2</td>
<td>0.5</td>
<td>0.33</td>
</tr>
<tr>
<td></td>
<td>0.55</td>
<td>0.35</td>
<td>0.63</td>
<td>0.31</td>
<td>0.46</td>
<td>0.68</td>
<td>0.35</td>
<td>0.38</td>
</tr>
<tr>
<td></td>
<td>0.16</td>
<td>0.24</td>
<td>0.35</td>
<td>0.15</td>
<td>0.24</td>
<td>0.24</td>
<td>0.18</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>0.55</td>
<td>0.45</td>
<td>0.68</td>
<td>0.31</td>
<td>0.45</td>
<td>0.68</td>
<td>0.37</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.47</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
<td>1.1</td>
<td>0.53</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>0.55</td>
<td>0.34</td>
<td>0.89</td>
<td>0.55</td>
<td>0.34</td>
<td>0.89</td>
<td>0.33</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.08</td>
<td>0.38</td>
<td>0.15</td>
<td>0.09</td>
<td>0.24</td>
<td>0.18</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>0.55</td>
<td>0.34</td>
<td>0.89</td>
<td>0.25</td>
<td>0.44</td>
<td>0.67</td>
<td>0.56</td>
<td>0.35</td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.08</td>
<td>0.38</td>
<td>0.15</td>
<td>0.09</td>
<td>0.24</td>
<td>0.18</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>0.55</td>
<td>0.34</td>
<td>0.89</td>
<td>0.25</td>
<td>0.44</td>
<td>0.67</td>
<td>0.56</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table II
EXECUTION TIMES FOR OUR REFINEMENT CHECKER USING DIFFERENT VARIABLE/VALUE SELECTION STRATEGIES FOR THE FOUR CAS VERSIONS. I STANDS FOR “FIND MUTATED ACTION”, 2 FOR “REACH & NON-REFINE”, AND TOTAL IS THE SUM THEREOF. ALL VALUES ARE GIVEN IN SECONDS.

<table>
<thead>
<tr>
<th>CAS version</th>
<th>leftmost-down 1: find mutated action 2: reach &amp; non-refine total</th>
<th>Ulysses in/out</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS_1</td>
<td>9 18 27</td>
<td>18 41</td>
</tr>
<tr>
<td></td>
<td>0.04 0.09 0.13</td>
<td>0.18 0.09 0.27</td>
</tr>
<tr>
<td></td>
<td>0.55 0.35 0.63</td>
<td>0.35 0.38 0.56</td>
</tr>
<tr>
<td></td>
<td>0.16 0.24</td>
<td>0.18 0.09 0.27</td>
</tr>
<tr>
<td></td>
<td>0.55 0.45 0.68</td>
<td>0.37 0.37 0.61</td>
</tr>
<tr>
<td></td>
<td>0.25 0.47</td>
<td>0.53 0.51 0.89</td>
</tr>
<tr>
<td></td>
<td>0.55 0.34</td>
<td>0.33 0.62 0.84</td>
</tr>
<tr>
<td></td>
<td>0.25 0.08</td>
<td>0.15 0.09 0.24</td>
</tr>
<tr>
<td></td>
<td>0.55 0.34</td>
<td>0.33 0.62 0.84</td>
</tr>
</tbody>
</table>

a timeout of one hour. Hence, the normalisation of action systems and the application of the one-point rule is really needed to achieve this essential performance gain.

E. Mutation Detection Strategies
Although the above results are already promising, they can be further improved. As can be seen in Table II, the time needed for finding the mutated action (I) still takes a considerable amount of time (33 - 97% of the total time needed for refinement checking). As already proposed in Section IV-C, a syntactic analysis to find the mutated action should solve this problem. So far, mutation detection has been realised by passing our non-refinement constraints to the constraint solver. This has the advantage that only “real” semantical mutations are detected, but costs its price in terms of long runtimes. A simpler and faster way to identify which action has been mutated is to perform mutation detection on a syntactic level, i.e. by comparing the source code of the actions. Table III lists the execution times in seconds needed for syntactic mutation detection “ff” for our four CAS versions and the six combinations of variable and value selection strategies. Syntactic mutation detection leads to runtimes that are drastically decreased compared to semantical mutation detection using the constraint solver. For each CAS version and for each combination of variable/value selection strategies, the time to find the mutated action for all 207 mutants is below one second. Hence, the total time needed for refinement checking now basically consists of the time needed for the combined reachability and non-refinement check. For this Step “2” of our process, we have omitted the runtimes in Table III as they are almost the same as in Table II. Through syntactic mutation detection, we achieved runtimes of 19 - 23 seconds to process all 207 mutated models for each version of the CAS. Hence, the settings for the constraint solver on how to choose variables and values have become irrelevant. We now clearly outperform the explicit ioco checker Ulysses, for which the runtimes to process the 207 mutated models have already been given in Table I. Ulysses needs 103 and 68 seconds depending on the distinction between input and output actions. Already for CAS_10, it exceeds its limits and needs around 8 hours.

VI. RELATED WORK
To our knowledge, this is the first test case generation approach 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
Rule. The experiment with the Car Alarm System shows that disjunctive normal form transformation and the one-point systems: Firstly, we implemented quantifier elimination via time of our refinement checker for non-deterministic action relying on explicit state space enumeration. (qualitative) action systems [18]. Both are not symbolic, but mance [19]) checkers for LOTOS specifications [30] and ing non-determinism include two ioco (input-output confor-

testing non-determinism: modular model checking of a composition test cases. However, this approach is not mutation-based. Refinement) for the CSP process algebra [29] to generate checker/refinement checker for FDR (Failures-Divergence 
[28] also considers non-determinism. It uses the model 
checkers deal with deterministic systems. Neverthe-

distinguish test cases. Gotlieb et al. do not use mutations, generation approaches on the source code level, where no non-determinism has to be considered. A mutation-based approach is [20] 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 [21], they work with constraint solving, in [22] with CLP. Regarding black-box techniques, one of the first models to be mutated were predicate-calculus specifications [23] and formal Z specifications [24]. 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 [25]. Most test case generation approaches using model checkers deal with deterministic systems. Nevertheless, there also exist works considering non-determinism and the involved difficulties. [26] suggests to synchronise non-deterministic choices in the original and the mutated model via common variables to avoid false positive counterexamples. [27] 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. [28] also considers non-determinism. It uses the model checker/refinement checker for 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 [19]) checkers for LOTOS specifications [30] and (qualitative) action systems [18]. Both are not symbolic, but rely on explicit state space enumeration.

VII. CONCLUSION

We presented three techniques to decrease the execution time of our refinement checker for non-deterministic action systems: Firstly, we implemented quantifier elimination via disjunctive normal form transformation and the one-point rule. The experiment with the Car Alarm System shows that we can now deal with all 207 mutants and that the execution time could be reduced to 41 seconds for the simplest model. However, for larger time parameters the tool still runs for 4.2 hours. Secondly, we experimented with different search strategies of the constraint solver. The best strategy reduced the execution time to 27 seconds. However, there is a great variety and it is hard to decide which strategy works best for which model. We noticed that the semantic check for finding the mutated action causes this variety and, thirdly, added a syntactic search leading to almost constant execution time for all models and all search strategies. As a result, the refinement checker is able to check all 207 mutants in 19 seconds. This is a performance gain of 80%.

The main contribution of this paper is to present and to evaluate the techniques that gained this performance win. From a broader perspective, this shows that non-determinism can be dealt with. Today, most commercial model-based test case generators exclude non-determinism. This line of research aims for bringing model-based mutation testing into practice — including non-determinism.

We are aware that one Car Alarm System example is not sufficient to generalise. Therefore, we are in the process of conducting additional experiments. Larger models with more variables will show the limitations. We plan to counter them with compositional test case generation and parallelisation. As with all undecidable problems, the aim is to push the limits until they become theoretical limitations.

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


