Towards Symbolic Model-Based Mutation Testing: Pitfalls in Expressing Semantics as Constraints

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

Abstract—Model-based mutation testing uses altered models to generate test cases that are able to detect whether a certain fault has been implemented in the system under test. For this purpose, we need to check for conformance between the original and the mutated model. We have developed an approach for conformance checking of action systems using constraints. Action systems are well-suited to specify reactive systems and may involve non-determinism. Expressing their semantics as constraints for the purpose of conformance checking is not totally straightforward. This paper presents some pitfalls that hinder the way to a sound encoding of semantics into constraint satisfaction problems and gives solutions for each problem.

Index Terms—action systems; model-based testing; mutation testing; conformance; refinement

I. INTRODUCTION

In recent years, model-based testing has become a well-known and established methodology for automatic test case generation. A formal model of the system under test (SUT) allows to automatically create test cases and serves as a test oracle. In order to derive test cases from the test model, some kind of test criterion is required. It provides information about what shall be tested. Examples for test criteria are manifold. Often, some coverage criterion or random traversal on the test model are used. They do not involve extra effort, but do not systematically cover functionality. Some strategies use test purposes stating that for example a specific transition or sequence of transitions shall be traversed. A disadvantage thereof is that they have to be written manually.

We follow a fault-centred approach, i.e., use mutations for test case generation. Classical mutation testing assesses the quality of an existing test suite. The program’s source code is altered by mutation operators and existing test cases are executed on the resulting mutants. If not at least one test case is able to kill a mutant, the test suite has to be improved or the mutant does not behave differently (equivalent mutant). For more details on mutation testing, we refer to the survey of Jia and Harman [1]. We employ the mutation concept on the test model instead of the source code and generate test cases that are able to kill the mutated models if they are not equivalent (model-based mutation testing). The generated tests are then run on the SUT and will detect whether a modelled fault has been implemented.

To model the SUT, we use Back’s action systems [2], [3]. In [4], we presented a symbolic approach for refinement checking of action systems via constraint solving techniques to avoid state space explosion. Interesting aspects are that we deal with reactive systems and allow non-determinism. Reactive systems make it necessary to perform a reachability analysis in order to find out whether non-conformance may actually occur. We use a predicative semantics of action systems to encode the transition relation and use it for the reachability analysis. For each reached state, we test whether it fulfills a constraint system that represents the conformance relation. For non-deterministic systems, refinement is a suitable conformance relation. We have implemented our refinement checker for action systems in SICStus Prolog1.

This paper gives details on the constraint solving approach that have not been discussed in [4]. It focuses on the conformance relation and on the predicative semantics of action systems. More precisely, it highlights pitfalls that we identified during the development of our framework. We use examples to demonstrate what does not work and present solutions for each pitfall.

This paper is organized as follows: Section II introduces action systems and Section III deals with conformance and non-determinism. Section IV presents pitfalls when developing a semantics of action systems for conformance checking. Section V considers to use Constraint Logic Programming (CLP). Our solutions are demonstrated on a small example in Section VI. Finally, Section VII concludes the paper.

II. ACTION SYSTEMS

Our chosen modelling formalism are action systems [2], which are well-suited to model reactive and concurrent systems. They have a formal semantics with refinement laws and are compositional [3]. Many extensions exist, but the main idea is that a system state is updated by guarded actions that may be enabled or not. If no action is enabled, the action system terminates. If several actions are enabled, one is chosen non-deterministically. Hence, concurrency is modelled in an interleaving semantics.

There exist various versions of Back’s original action system notation [2]. 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

1http://www.sics.se/sicstus
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 \( \vec{X} \). 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 \([\ldots]\). 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.

A. Car Alarm System Example

As an example consider the following car alarm system (CAS): It is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed. The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes. The CAS can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

Listing 1 presents a simplified, exemplary code snippet from an action system modelling the CAS. 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 an action system modelling the CAS. 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 and sound are turned on. Line 3 defines the list of variables that make up the state of the action system. The \( \text{init} \) predicate in Line 4 defines the initial values for the state. The \( \text{actions} \) block (Lines 6 to 16) defines named actions, which consist of a name, a guard and a body (\( \text{name} :: \text{guard} => \text{body} \)). The action ‘AlarmOn’ (Lines 7 to 14) models the activation of the alarm, i.e., the triggering of the blinking flash lights and the sound. It is not specified in which order these two alarms are turned on. This is modelled by the sequential composition \( (\ldots) \) of two non-deterministic choices \([\ldots]\). Lines 8 to 10 non-deterministically either activate the flash lights \( f \) or the sound \( s \). Subsequently, the other alarm (the one which is not yet enabled) is turned on. The do-od block (Line 17) 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 distinguish two action systems. We accomplish this by performing a conformance check between the original and the mutant. Counterexamples are witnesses for non-conformance and represent test cases that reveal the injected faults. The comment \( % \) in Line 8 of Listing 1 represents a possible mutant for the CAS. 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 CAS activates two alarms (flash and sound) in the action \( \text{AlarmOn}, \) i.e., it sets the variables \( f \) and \( s \) to true. The mutated action system cannot always establish this behaviour. Although it can establish the right post-state by first activating the sound (Line 10) and then enabling the flash lights in Line 12, it also might end up in a wrong post-state by first executing the mutated statement of Line 8. Afterwards, both branches of the second non-deterministic choice are enabled. In case of choosing Line 12 the flash lights will be turned on, but no sound. If Line 14 is executed, the sound will be enabled, but no flash lights. These two scenarios are counterexamples to conformance. They attest non-conformance and allow us to derive test cases that check whether the modelled fault has been implemented in a SUT.

III. CONFORMANCE RELATION

As just explained, we want to generate test cases that are able to distinguish two systems (a specification and an implementation). For this purpose, we need to check for conformance. In our case, we have to find out whether a mutated model conforms to the original one. Our first pitfall affects the choice of an appropriate conformance relation.

**Pitfall 1.** For deterministic systems, non-equivalence checking is a standard approach for finding counterexamples to conformance that allow the derivation of distinguishing test cases [5]. However, it is not suitable when non-determinism is involved (as it is the case for action systems). Just assuming the same inputs and asserting that at least one output of the mutated system differs from the output of the original one is not sufficient and leads to wrong results. Consider the following example.

**Example 1.** A system returns either 1 or 2 as an output regardless of the input. This can be expressed by the constraints \( C^o \):

\[
C^o = (\text{out}^o = 1 \lor \text{out}^o = 2)
\]

The variable \( \text{out}^o \) represents the return value of this original system. A mutated version could return 2 or 3. The variable \( \text{out}^m \) represents the output of the mutated system in the corresponding constraints \( C^m \):

\[
C^m = (\text{out}^m = 2 \lor \text{out}^m = 3)
\]

By requiring different outputs we get \( C^o \land C^m \land \text{out}^m \neq \text{out}^o \).
There exist three solutions satisfying these constraints: (1) \( out^m = 2, out^o = 1 \), (2) \( out^m = 3, out^o = 1 \), and (3) \( out^m = 3, out^o = 2 \). Obviously, the second and the third are real counterexamples, since the mutant returns 3 which is not specified by the original system - neither in the first nor in the second branch. The first solution is not a real difference between the two systems. Return value 1 is not the only specified output. Also 2 is allowed by the original. Hence the mutant does nothing wrong if it returns 2.

**Example 2.** The problem becomes even more obvious if we check for equivalence between a specification and itself. In this case, we should not find any counterexample for conformance, since each system conforms to itself. Again, consider the system’s specification \( C^o \) from above and the very same specification as implementation. The following constraints encode non-equivalence:

\[
C^o = (out^o = 1 \lor out^o = 2) \\
C^m = (out^m = 1 \lor out^m = 2) \\
C^o \land C^m \land out^m \neq out^o
\]

There exist solutions for these constraints \( (out^m = 2, out^o = 1) \) and \( (out^m = 1, out^o = 2) \), which is not what we expect. Hence, for non-deterministic systems equivalence checking leads to false positive counterexamples.

These examples illustrate that an equivalence relation assuming same inputs and at least one different output is not a suitable conformance relation for non-deterministic systems. 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 [6] giving \( M \) and \( I \) a predicative semantics.

**Definition 1. (Refinement)**

\[
M \sqsubseteq I \iff \forall x, x', y, y', \cdots \in \alpha : I \Rightarrow M
\]

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.

We already developed a mutation testing theory built upon this notion of refinement [7]. It is based on the idea 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 searching for counterexamples to refinement. From Definition 1 follows that such counterexamples exist if and only if implication does not hold:

\[
\exists x, x', y, y', \cdots \in \alpha : M^m \land \neg M^o
\]

This formula expresses that there are observations in the mutant \( M^m \) that are not allowed by the original model \( M^o \).

**Example 3.** Let’s reconsider the above examples. For our refinement relation we now get the following constraints:

\[
C^m \land \neg C^o \land out^m = out^o
\]

By applying \( C^o \) and \( C^m \) from Example 1, we now get only one solution \( (out^m = out^o = 2) \), which is the only correct counterexample. Considering \( C^o \) and \( C^m \) from Example 2, we get no solution any more. This reflects what we want since we compared two identical systems.

**IV. Semantics**

So far, we have not formally defined the semantics of action systems. This also has some pitfalls in store as will be discussed in the following.

For encoding deterministic programs as constraints, it is a practical way to use static single assignment (SSA) form [8]. The SSA form is an established intermediate representation for programs, where each variable is defined only once. This is accomplished by introducing a new identifier for each variable on the left-hand side of an assignment. This works fine for the positive case, i.e., for the exploration of the state space of a system, but entails problems when negation is required as in model-based mutation testing using refinement (cf. Definition 1).

**Example 4.** Consider the following specification using sequential composition and its constraint representation commonly used, e.g., in [5].

\[
\begin{align*}
out := 1; & \quad out := out + 1 \\
C^o = (out^o_1 = 1 \land out^o_2 = out^o_1 + 1)
\end{align*}
\]

A possible implementation could be:

\[
\begin{align*}
out := 2; & \quad C^m = (out^m = 2)
\end{align*}
\]

Since both systems return 2 in any case, refinement holds and we should not find a counterexample. Non-refinement is expressed by the following constraints:

\[
C^m \land \neg C^o \land out^m = out^o_2
\]
Unfortunately, these constraints can be satisfied by $out^m = out^2_1 = out^2_2 = 2$, which wrongly classifies our implementation $out := 2$ being incorrect.

**Pitfall 2.** The above example demonstrates that there is a problem with specifications comprising sequential composition. Constraints for such specifications that are derived via the SSA form cannot be easily negated as it results in false positives even in the deterministic case.

The problem is that the pure SSA form does not reflect the semantics completely correctly. Typically, the formal semantics of action systems are defined in terms of weakest preconditions. However, for our constraint-based approach we propose the use of a relational predicative semantics that follows the style of UTP [6]. This approach has already been used for relations (without sequential composition) [9], [10] and a guarded command language similar to action systems [7].

Figure 2 presents the formal predicative semantics for actions of our modelling language (Figure 1). State changes are defined via predicates relating the pre-state of all variables $\pi$ and their post-state $\pi'$. Moreover, the labels of actions form a visible trace of events $tr$ that is updated to $tr'$ whenever an action terminates successfully. Hence, a guarded action’s transition relation is defined as the conjunction of its guard $g$, its body $B$ and the construction of the post-trace by adding the action’s label $l$ to the previous trace. If the action has parameters $\overline{X}$, they are added as local variables to the predicate. Assignments update one variable $x$ with the value of an expression $e$ and preserve the remaining variables unchanged.

The main difference to our semantics using SSA form is in sequential composition. While sequential composition in SSA form was too simply defined as disjunction, in UTP it is defined using existential quantification of the intermediate variables: there must exist an intermediate state $\overline{m}$ 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.

The semantics via SSA form and via UTP are solely unequal when it comes to negation. In the positive case, the SSA form for sequential composition is $B(\pi, \overline{m}) \land B(\overline{m}, \pi')$. Existential quantification of $\overline{m}$ is done implicitly by the constraint solver. This makes it equal to our predicative semantics for sequential composition, which is $\exists \overline{m} : (B(\pi, \overline{m}) \land B(\overline{m}, \pi'))$. If we negate the latter we get $\neg (\exists \overline{m} : (B(\pi, \overline{m}) \land B(\overline{m}, \pi')))$. If we negate the SSA form, we get $\neg (B(\pi, \overline{m}) \land B(\overline{m}, \pi'))$.

Again, the constraint solver implicitly existentially quantifies all variables and we get $\exists \overline{m}, \overline{n}, \overline{r} : (\neg B(\pi, \overline{m}) \land B(\overline{m}, \pi'))$, which is wrong as the existential quantification is not negated. Hence the source of Pitfall 2 is a wrongly placed existential quantification. This leads us directly to the next pitfall.

**Pitfall 3.** Our predicative semantics lead to the following constraints for a negated sequential composition:

$\neg (\exists \overline{m} : (B(\pi, \overline{m}) \land B(\overline{m}, \pi'))) \land \neg (\exists \overline{m} : (B(\pi, \overline{m}) \land B(\overline{m}, \pi')))$

By resolving negation, we get

$\forall \overline{m} : (\neg (B(\pi, \overline{m}) \land B(\overline{m}, \pi'))) \land (\exists \overline{m} : (B(\pi, \overline{m}) \land B(\overline{m}, \pi')))$

This constraint system expresses exactly what is intended, but uses universal quantification. Universal quantification can only be expressed in quantified constraint satisfaction problems (QCSPs), which are not supported by common constraint solvers.

Based on the original constraints for a negated sequential composition, we are able to resolve this problem by application of the so-called one-point rule:

$$(\exists x : e = e \land P(x)) \Leftrightarrow P(e)$$

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.

**Pitfall 4.** 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 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 rewritten to this normal form. We have implemented this transformation in Prolog.
V. CONSTRAINT LOGIC PROGRAMMING

As already presented in Figure 2, the semantics of an action also comprises its action label. Hence, we also have to encode action names (labels) and their parameters. At the moment we use SICStus Prolog’s built-in constraint solver [11]. As it only supports integers, each action and its parameters has to be associated with an integer. Constraint logic programming combines logic programming with constraint solving. In this way, it becomes possible to encode actions and their parameters as Prolog terms and use unification to compare them. This facilitates the treatment of actions, but also leads us to another pitfall.

Pitfall 5. All operators dealing with operands that contain Prolog clauses must be Prolog operators, not operators from the constraint solver. Conjunction in Prolog expressed by a comma ‘,’; negation by ‘\-’. Hence, our constraints to find a counterexample

\[ \exists x, x', y, y' \in \alpha : M^m \land \neg M^o \]

would have to be rewritten into

\[ \exists x, x', y, y', \ldots \in \alpha : M^m, \neg M^o \]

Unfortunately, Prolog’s negation is not equivalent to logical negation as is well known. Prolog implements negation as failure, i.e., ‘\+P’ means that \(P\) is not provable, whereas ‘\neg P’ means \(P\) is not true.

Example 5. Again, consider the specifications introduced in Example 1. From Example 3, we know the following counterexample: \(out^m = out^o = 3\). If we use Prolog’s negation as described above, we get the following CLP problem (all domains range from 1 to 3):

\[ out^m = out, \ (out^m = 2 \lor out^m = 3), \ \neg( out = 1 \lor out = 2) \]

Prolog’s answer for this query is no, i.e., it cannot find a counterexample. The reason is that the variables \(out^m\) and \(out\) are not instantiated before the negated term. They are just fixed to be 1 or 2. In this case, Prolog can prove \(out = 1 \lor out = 2\) as the right-hand side of the disjunction is true for \(out = 2\). As the goal \(out = 1 \lor out = 2\) is provable, the negated goal fails. Hence, the whole CLP goal fails and no counterexample is found.

The problem with negation as failure could be avoided by instantiation of all variables before the call of negation. This can be established by letting the constraint solver enumerate all possible values for the variables. Prolog’s backtracking would then ensure that all possibilities are tested. This solution corresponds to other techniques that use explicit enumeration, which suffer from state space explosion. In [4], we used the full car alarm system to indicate that these techniques are inferior to our symbolic approach. For this reason we suppose to use pure constraint solving techniques and encode actions and their parameters as integers.

VI. DEMONSTRATION OF CORRECT SOLUTION

In the following we demonstrate our refinement checking framework that resolves all of the presented pitfalls using the CAS introduced in Section II-A. The first step is to normalize the action system depicted in Listing 1 to avoid Pitfall 4. Remember that the normal form corresponds to DNF and requires that non-deterministic choice is always the outermost operator and not allowed in nested expressions. Normalization has to be applied to each action’s body as this is the only place where our syntax allows a combination of non-deterministic choice and sequential composition (cf. Figure 1). Listing 2 shows the normalization of the action \(AlarmOn\), which is a non-deterministic choice of four sequential compositions.

Next, the action must be encoded as constraints. By applying our predicative semantics of Figure 2, we avoid Pitfall 2. The resulting constraints are depicted in Figure 3 assuming that the integer encoding for label \(AlarmOn\) is 1. Note that we cannot encode a whole trace of actions in the constraints, but use a variable \(act'\) to get the executed action (cf. Equation 6) and then concatenate the trace in Prolog. This is possible since each iteration of the do-od block may execute at most one action. By simplification, we can skip Equation 1. Equations 2 and 5 are part of a disjunction and eliminated since they are contradictions. By application of the one-point rule, we avoid Pitfall 3. Altogether, this results in the following constraints:

\[ ((f = 0 \land s = 0 \land f' = 1 \land s' = 1) \lor (s = 0 \land f = 0 \land f' = 1 \land s' = 1)) \land act' = 1 \]

Both cases of the disjunction are equivalent and may be reduced to one, which will be referred to as \(C^o\) in the following:

\[ C^o = (f = 0 \land s = 0 \land f' = 1 \land s' = 1 \land act' = 1) \]
This expresses what was intended to be modelled: the action `AlarmOn` (encoded by 1) is executed if neither sound nor flash are activated yet and turns on both alarms.

Working analogously for the mutant added as a comment in Line 8 of Listing 1, we get the constraints \( C^m \):
\[
C^m = ((f^m = 0 \land f^{m'} = 1 \land s^{m'} = s^m) \lor
(f^m = 0 \land s^m = 0 \land f^{m'} = 0 \land s^{m'} = 1) \lor
(s^m = 0 \land f^m = 0 \land f^{m'} = 1 \land s^{m'} = 1)) \land
act^{m'} = 1)
\]
(7)\(\) (8)\(\) (9)\(\) (10)

Therefore, the refinement constraints are (domains and initial values are omitted for the sake of simplicity):
\[
C^m \land \neg C^0 \land f^m = f \land s^m = s \land ... \land act^{m'} = act'
\]
The constraint solver provides two solutions: (1) \( f^f = f^{m'} = 0, s^f = s^{m'} = 1 \) and (2) \( f^f = f^{m'} = 1, s^f = s^{m'} = 0 \). The second solution represents Equation 7 of \( C^m \), the first solution corresponds to Equation 8. Both solutions serve as counterexamples for refinement as they reveal wrong behaviour. They do not set both alarms (flash and sound) to true in the post-state. Note that all initial values are instantiated to 0 and that the action variables \( act' \) and \( act^{m'} \) are always 1.

If we perform a non-refinement check of the original with itself, we do not get any solution. This is what we want as every systems refines itself.

VII. CONCLUSION

In this paper, we have shown pitfalls in the definition of a proper conformance relation and semantics for action systems in the context of model-based mutation testing. For each pitfall, we presented a solution that made our mutation-based test case generation approach 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. Regarding model-based mutation testing, the following works are closely related: [5] uses the SSA form of Java-like programs to express their semantics and generate distinguishing test cases. Model checkers work very similar to our approach. They check for equivalence of temporal formulae. In case of non-equivalence, counterexamples serve as test cases [12]. Gotlieb et al. do not use mutations, but structural criteria for test data generation via SSA form. In [13], they work with constraint solving, in [14] CLP is used. However, these works do not consider non-determinism. Model-based mutation testing techniques considering non-determinism include two ioco (input-output conformance) checkers for LOTOS specifications [15] and (qualitative) action systems [16]. Both are not symbolic, but rely on explicit state space enumeration. FDR (Failures-Divergence Refinement) for the CSP process algebra [17] considers non-determinism, too. The corresponding FDR model checker/refinement checker has been used for test case generation in [18]. However, this approach is not mutation-based.

We have various ideas for future work. First steps include the use of SMT solvers for which most pitfalls are still relevant. Nevertheless, they seem to be promising as there exists a theory that supports quantifiers. Thereby, Pitfall 3 would become pointless.

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