Formal mutation testing for Circus

Alex Alberto, Ana Cavalcanti, Marie-Claude Gaudel, Adenilso Simao

Formal mutation testing for *Circus*

Alex Alberto[a], Ana Cavalcanti[b], Marie-Claude Gaudel[c], Adenilso Simão[a]

[a] *Universidade de São Paulo, ICMC, São Carlos, Brazil*
[b] *University of York, Department of Computer Science, York YO10 5GH, UK*
[c] *LRI, Université de Paris-Sud and CNRS, Orsay 91405, France*

Abstract

Context: The demand from industry for more dependable and scalable test-development mechanisms has fostered the use of formal models to guide the generation of tests. Despite many advancements having been obtained with state-based models, such as Finite State Machines (FSMs) and Input/Output Transition Systems (IOTSs), more advanced formalisms are required to specify large, state-rich, concurrent systems. *Circus*, a state-rich process algebra combining Z, CSP and a refinement calculus, is suitable for this; however, deriving tests from such models is accordingly more challenging. Recently, a testing theory has been stated for *Circus*, allowing the verification of process refinement based on exhaustive test sets.
Objective: We investigate fault-based testing for refinement from *Circus* specifications using mutation. We seek the benefits of such techniques in test-set quality assertion and fault-based test-case selection. We target results relevant not only for *Circus*, but to any process algebra for refinement that combines CSP with a data language.
Method: We present a formal definition for fault-based test sets, extending the *Circus* testing theory, and an extensive study of mutation operators for *Circus*. Using these results, we propose an approach to generate tests to kill mutants. Finally, we explain how prototype tool support can be obtained with the implementation of a mutant generator, a translator from *Circus* to CSP, and a refinement checker for CSP, and with a more sophisticated chain of tools that support the use of symbolic tests.
Results: We formally characterise mutation testing for *Circus*, defining the exhaustive test sets that can kill a given mutant. We also provide a technique to select tests from these sets based on specification traces of the mutants. Finally, we present mutation operators that consider faults related to both reactive and data manipulation behaviour. Altogether, we define a new fault-based test-generation technique for *Circus*.
Conclusion: We conclude that mutation testing for *Circus* can truly aid making test generation from state-rich model more tractable, by focussing on particular faults.

Keywords: *Circus*, mutation, testing, formal specification

1. Introduction

Testing from formal models is currently advancing as a solid approach to support the growing demand from industry for more dependable and scalable test-development

mechanisms. For instance, Model-Based Testing (MBT) benefits greatly from a precise and clear semantics for models, as opposed to informal or semi-formal models whose semantics is dependent on the particular tool in use.

Many advancements have been obtained with state-based models, such as Finite State Machines (FSMs) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] and Input/Output Transition Systems (IOTSs) [11, 12, 13, 14, 15]. Those models, however, quickly become intractable when dealing with larger systems. Thus, more advanced formalisms are required to facilitate the specification of large, state-rich, concurrent systems.

Circus is a state-rich process algebra combining Z [16], CSP [17], and a refinement calculus [18]. Its denotational and operational semantics are based on the Unifying Theories of Programming (UTP) [19]. *Circus* can be used to verify large concurrent systems, including those that cannot be handled by model checking. *Circus* has already been used to verify, for example, software in aerospace applications [20], and novel virtualization software by the US Naval Research Laboratory [21].

A theory of testing for *Circus* [22], instantiating Gaudel's long-standing theory of formal testing [23, 24, 25], is available. It is founded on the *Circus* operational semantics [26], described and justified in the UTP [27]. As usual in testing, it considers divergence-free processes for the model and the system under test. More precisely, if a system under test diverges, since one cannot decide whether it is deadlocked or divergent, divergence is assimilated to an unspecified deadlock and detected as a failure.

The *Circus* testing theory introduces potentially infinite (symbolic) exhaustive test sets. To achieve practical usefulness, it is, therefore, mandatory to rely on selection criteria both to generate and to select a finite set of tests.

Test-case generation in model-based testing is guided by testing requirements that should be met by a test suite. Usually, the requirements are either coverage criteria that state which elements of the model should be traversed (covered) by test execution, or fault models, which define specific faults that the test cases are supposed to reveal, if present in the system. These approaches are usually complementary to each other. Coverage-based testing is proposed for *Circus* in [28]. It is worth investigating how fault-based testing can complement the coverage testing.

Mutation testing is recognized as one of the most effective fault-detection techniques [29]. The systematic injection of feasible modeling faults into specifications allows the prediction of potential defective implementations. The faults are seeded by syntactic changes that may affect the observable specified behavior. Such faulty models are "mutants". A mutant is "killed" by a test case able to expose its observable behavior difference. Testing can benefit from mutation in two ways [30]: some quality aspects of a test set can be measured by the number of mutants it can kill, and the analysis of a mutant model allows the selection of tests targeting specific faults or fault classes.

In this paper, we introduce an approach to apply mutation testing to *Circus* specifications. Most of the presented mutation operators, that is, the fault-injection strategies, are based on previous works that have tackled similar challenges in related modeling languages [31, 32, 33]. The outcome of all mutation operators are, however, analyzed considering the specific features and particularities of *Circus*. Moreover, our results are valid in the context of other process algebras, especially those based on CSP [34, 35].

The contribution of this paper is manifold. First, we instantiate the notions of mutation testing for a state-rich concurrent language, namely, *Circus*, and its formal

theory of testing. In particular, we face the challenge of associating mutations in the text of a *Circus* specification to traces of the *Circus* denotational semantics that define tests that cover the mutation. Even though mutation testing has already been applied to languages and theories upon which *Circus* is based, such as CSP [31] and the UTP [36], the consideration of a state-rich process algebra for refinement with a UTP semantics is novel. Second, we propose mutation operators for *Circus*, analysing and adapting existing ones for the underlying languages and designing some that are specific to *Circus*. Third, we describe prototype tool support for the application of the mutant operators and two approaches to generate tests that can kill these mutants. When it is feasible to translate the considered *Circus* specification into CSP, we propose the use of the FDR model checker. For the other cases, we identify a tool chain that copes directly with *Circus* specifications via slicing techniques and symbolic execution.

This paper is organized as follows. Section 2 gives an overview of the aspects of *Circus* and its testing theory that we use here. Section 3 extends the testing theory to consider mutation testing and describes our approach to generating tests based on mutants. The mutation operators used to generate the mutants themselves are defined in Section 4. Tool support for automation of our approach is discussed in Section 5, and an extra complete example is introduced in Section 6. Finally, we present some related and future work and conclusions in Sections 7 and 8.

2. *Circus* and its testing theory

In this section, we give a brief description of the *Circus* language, its operational semantics [26], and its testing theory [22].

2.1. Circus notation and operational semantics

As exemplified is Figure 1, *Circus* allows us to model systems and their components via (a network of) interacting processes. In Figure 1, we define a single process $Chrono$ that specifies the reactive behaviour of a chronometer. This is a process that recognises $tick$ events that mark the passage of time, a request to output the current time via a channel $time$, and outputs minutes and seconds via a channel out.

A *Circus* specification is defined by a sequence of paragraphs. Roughly speaking, they define processes, but also channels, and any types and functions used in the process specifications. In Figure 1, we define a type $RANGE$, including the valid values for seconds and minutes, and the channels $tick$, $time$ and out. The channel out is typed, since it is used to communicate the current minutes and seconds recorded in the chronometer as a pair. The final paragraph in Figure 1 defines $Chrono$ itself.

Each process has:

a state and some operations for observing and changing it in a Z style. In $Chrono$, the state is composed by a pair $AState$ of variables named sec and min with integer values between 0 and 59 (as defined by $RANGE$), and the data operations on this state are specified by the three schemas $AInit$, $IncSec$, $IncMin$.

actions that define communicating behaviours in a CSP style. The overall behaviour of a process is specified by the main action after the symbol $\bullet$. In our example, it is a

$$
\begin{array}{l}
RANGE == 0 \mathinner{..} 59 \\
\textbf{channel}\ tick, time \\
\textbf{channel}\ out : RANGE \times RANGE \\
\\
\textbf{process}\ Chrono \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ AState == [\, sec, min : RANGE \,] \\
\qquad AInit == [\, AState' \mid sec' = min' \land min' = 0 \,] \\
\qquad IncSec == [\, \Delta AState \mid sec' = (sec + 1) \bmod 60 \land min' = min \,] \\
\qquad IncMin == [\, \Delta AState \mid min' = (min + 1) \bmod 60 \land sec' = sec \,] \\
\qquad Run \mathrel{\widehat=} tick \rightarrow IncSec;\ ((sec = 0)\ \& \ IncMin) \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad\qquad\qquad ((sec \neq 0)\ \& \ \textbf{Skip}) \\
\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad time \rightarrow out\,!(min, sec) \rightarrow \textbf{Skip} \\
\qquad \bullet\ (AInit;\ (\mu\ X \bullet (Run;\ X))) \\
\textbf{end}
\end{array}
$$

Figure 1: A *Circus* specification of a chronometer

sequential composition of the schema $AInit$ followed by the repeated execution of the Run action. The *Circus* construct $\mu\ X \bullet A(X)$ defines a recursive action A, in which X is used for recursive calls.

The initialisation schema $AInit$ defines the values sec' and min' of the state components after the initialisation. These components are declared using $AState'$. The operation schemas $IncSec$ and $IncMin$ change the state, as indicated by the declaration $\Delta AState$. They also define values sec' and min' of the state components after the operations. In each case, the seconds and minutes are incremented modulo 60.

Run starts with an external choice ($\Box$) between the events $tick$ and $time$. If the environment chooses the event $tick$, this is followed by the increment of the chronometer using the data operation $IncSec$. Afterwards, we have another choice between actions guarded by the conditions $sec = 0$ and $sec \neq 0$. If, after the increment, we have $sec = 0$, then the minutes are incremented using $IncMin$. Otherwise, the action terminates (**Skip**). If the event $time$ occurs, then the values of min and sec are displayed (output), using the channel out, before termination.

Circus comes with a denotational and an operational semantics, based on Hoare and He's Unifying Theories of Programming (UTP) [19], and a notion of refinement. We can use *Circus* to write abstract as well as more concrete specifications, or even programs. A full account of *Circus* and its denotational semantics is given in [37].

The operational semantics [26] plays an essential role on the definition of testing strategies based on *Circus* specifications. It is briefly introduced below, and a significant part is reproduced in Appendix A. It is defined as a symbolic labelled transition system between configurations. These are triples $(\mathtt{c} \mid \mathtt{s} \models \mathtt{A})$, with a constraint $\mathtt{c}$, a state $\mathtt{s}$, and a continuation $\mathtt{A}$, which is a *Circus* action. Transitions associate two con-

$$\frac{c \wedge T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x} : \mathtt{T} \to \mathtt{A}) \xrightarrow{\mathtt{d?w_0}} (\mathtt{c} \wedge \mathtt{w_0} \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\ x} := \mathtt{w_0} \models \mathtt{let\ x} \bullet \mathtt{A})}$$

$$\frac{c \wedge (s;\ g)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{g} \,\&\, \mathtt{A}) \xrightarrow{\epsilon} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{g}) \mid \mathtt{s} \models \mathtt{A})}$$

Figure 2: Examples of two transition rules: for inputs, and for guards

figurations and a label. The labels are either empty, represented by ϵ, or symbolic communications of the form c?w or c!w, where c is a channel name and w is a symbolic variable that represents an input (?) or an output (!) value.

The first component c of a configuration ($\mathtt{c} \mid \mathtt{s} \models \mathtt{A}$) is a constraint over symbolic variables that are used to define labels and the state. The constraints are texts that denote *Circus* predicates over these symbolic variables. We use typewriter font for pieces of text. For example, $\mathtt{x} := \mathtt{w_0}$ is the text that describes an assignment of a value represented by a symbolic variable w_0 to the variable x. On the other hand, $x := w_0$ is the predicative relation that defines the meaning of $\mathtt{x} := \mathtt{w_0}$. The distinction is important in the operational semantics, which manipulates pieces of text to define constraints.

The second component s is a UTP predicate, which defines a total assignment $\mathtt{x} := \mathtt{w}$ of symbolic variables w to all variables x in scope, including the state components. State assignments, however, can also include declarations and undeclarations of variables using the constructs var x := e and end x. The state assignments define a value for all variables in scope. These values are represented by symbolic variables similarly to what is classically done in symbolic execution of programs [38].

Two examples of rules are given in Figure 2. The first rule defines the transitions arising from an input prefixing $\mathtt{d?x} : \mathtt{T} \to \mathtt{A}$; it is rule (A.3) of Appendix A. The label of the transition is $\mathtt{d?w_0}$, where w_0 is a symbolic variable. The constraint that w_0 is of the right type ($\mathtt{w_0} \in \mathtt{T}$) is added to the constraint of the new configuration. The state of the new configuration is enriched, via the UTP sequence operator "; ", by a new component x, which is assigned value w_0. The continuation of the new configuration is the action A in an environment enriched by x as defined by $\mathtt{let\ x} \bullet \mathtt{A}$.

The second rule of Figure 2 defines the transitions arising from a guarded action $\mathtt{g}\ \&\ \mathtt{A}$. The label of such a transition is empty, since the evaluation of g is not an observable event; g is added to the constraint of the new configuration taking into account the assignments in the current state s. The continuation is A.

Traces of a process are defined in the usual way, that is, as sequences of observable events. Due to the symbolic nature of configurations and labels, however, we can obtain from the operational semantics *constrained symbolic traces*, or *cstraces*, for short. These are pairs formed by a sequence of labels, that is, a symbolic trace, and a constraint over the symbolic variables used in the labels. Roughly speaking, the constrained symbolic trace can be obtained by evaluating the operational semantics, collecting the labels together, and accumulating the constraints over the symbolic

$$cst1 : (\langle tick \rangle, true)$$
$$cst2 : (\langle time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 0 \land \alpha_1 = 0)$$
$$cst3 : (\langle tick, time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 0 \land \alpha_1 = 1)$$
$$cst4 : (\langle tick \rangle^{60}, true)$$
$$cst5 : (\langle tick \rangle^{60} \frown \langle time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 1 \land \alpha_1 = 0)$$

Figure 3: Some constrained symbolic traces for $Chrono$

variables used in the labels. Figure 3 gives some examples of cstraces of $Chrono$.

A trace is an instantiation of a cstrace, where the symbolic variables used in the labels are replaced by values satisfying the constraint. For instance, the two traces $\langle time, out!0!0 \rangle$ and $\langle tick, time, out!0!1 \rangle$ are instances of $cst2$ and $cst3$.

2.2. Testing in Circus

Gaudel's long-standing testing theory [23] has been instantiated for *Circus* in [22]. The conformance relation considered in that work is process refinement: the UTP notion of refinement applied to state-rich processes.

As previously explained, the *Circus* testing theory takes the view that, in specifications, divergences are mistakes. In addition, since in a system under test (SUT), they are observed as deadlocks, altogether the results in [22] consider divergence-free specifications and SUT. For divergence-free models, the *Circus* refinement relation can be characterized by the conjunction of traces-refinement and the well known $conf$ relation, as defined in [39], that requires reduction of deadlocks. This is proved in [40].

Accordingly, [22] defines separate exhaustive test sets for traces refinement and $conf$, namely $Exhaust_T(SP)$ and $Exhaust_{conf}(SP)$, which we briefly present here.

A test for traces refinement is constructed by considering a trace of the *Circus* specification and one of the events that cannot be used to extend that trace to obtain a new trace of the *Circus* specification [41]. Such events are called the forbidden continuations of the trace. For a specification SP, the exhaustive test set $Exhaust_T(SP)$ includes all the tests formed by considering all the traces and all their forbidden continuations and inserting some special verdict events, as explained below.

For a finite trace $s = \langle a_1, a_2, \ldots, a_n \rangle$ and an event (forbidden continuation) a, we define the test process $T_T(s, a)$ as follows:

$$T_T(s, a) = inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow inc \ldots a_n \rightarrow pass \rightarrow a \rightarrow fail \rightarrow STOP$$

Extra special events inc, $pass$ and $fail$ are used to indicate a verdict. In the execution of a testing experiment, the test is run in parallel with the SUT and the last special event observed in a testing experiment provides the verdict. Due the possibility of nondeterminism, the submitted trace of the *Circus* specification is not necessarily performed by the SUT. The inc event indicates an inconclusive verdict: the SUT has not performed the proposed trace. If it does perform the trace, a $pass$ event is observed, but if the SUT proceeds to engage in the forbidden continuation a, then there is a $fail$ event.

Example 1. *For instance, a possible test of Chrono, based on the trace* $\langle tick, time \rangle$ *and the forbidden continuation* $out.0.0$ *is:*

$$inc \rightarrow tick \rightarrow inc \rightarrow time \rightarrow pass \rightarrow out.0.0 \rightarrow fail \rightarrow \textbf{Stop}$$

□

This leads to the following definition of the exhaustive test set for traces refinement:

$$Exhaust_T(SP) = \{\ T_T(s, a) \mid s \in traces(SP) \land s \frown \langle a \rangle \notin traces(SP)\ \}$$

The exhaustivity, that is, the equivalence of traces refinement to the absence of $fail$ verdict when running all the tests of $Exhaust_T(SP)$, is proved in [41] and [22] under – as usual in theoretical approaches to testing in the presence of nondeterminism – the *complete testing assumption* [42]: when a test experiment is performed a sufficient number of times all possible (nondeterministic) behaviours of the SUT are observed.

Traces and forbidden continuations are characterised symbolically leading to the definition of $SExhaust_T(SP)$, an exhaustive set of symbolic tests based on cstraces and constrained symbolic forbidden continuations.

Example 2. *An example of such a symbolic test for Chrono, based on the cstrace* $(\langle tick, time \rangle, true)$ *and the forbidden constrained symbolic continuation defined as* $out.\alpha_0.\alpha_1 : (\neg(\alpha_0 = 0 \land \alpha_1 = 1))$ *is as follows:*

$$inc \rightarrow tick : true \rightarrow inc \rightarrow time : true \rightarrow pass$$
$$\rightarrow out.\alpha_0.\alpha_1 : (\neg(\alpha_0 = 0 \land \alpha_1 = 1)) \rightarrow fail \rightarrow \textbf{Stop}$$

□

In [22] it is proved that $Exhaust_T(SP)$ corresponds to all the tests that are valid instances (that is, that satisfy the constraints) of some symbolic test in $SExhaust_T(SP)$.

The $conf$ relation captures reduction of deadlock. Given SP_1 and SP_2, we have that SP_1 $conf$ SP_2 if, and only if, whenever SP_2 engages in a sequence of events, that is, a trace that can be accepted by SP_1 as well, then SP_2 can only deadlock if SP_1 may as well. Formally, $conf$ can be defined as follows:

$$SP_2\ conf\ SP_1 \mathrel{\widehat{=}} \forall\, t : traces(SP_1) \cap traces(SP_2) \bullet Ref(SP_2, t) \subseteq Ref(SP_1, t)$$
where $Ref(SP, t) \mathrel{\widehat{=}} \{\ X \mid (t, X) \in failures(SP)\ \}$

For a trace t of a process P and a subset $X = \{a_1, \ldots, a_n\}$ of the set of events of P, noted αP, the pair (t, X) belongs to $failures(P)$ if, and only if, after performing t, P may refuse all events of X. In other words, the parallel composition below may deadlock just after t. We use $proc(t)$ to represent a *Circus* process that accepts just the execution of t before finishing; it can be defined using prefixing, for example.

$$P \llbracket\ \alpha P\ \rrbracket\ (proc(t);\ (a_1 \rightarrow P_1 \,\Box \ldots \Box\, a_n \rightarrow P_n))$$

$P \llbracket\ \alpha P\ \rrbracket\ Q$ is the parallel composition of the processes P and Q with synchronisation required on all the events of P, that is, the events in the set αP.

Thus, given a system under test SUT and a specification SP, for $SUT\ conf\ SP$ to hold, the definition requires that, after performing every one of their common traces, the failures of SUT are failures of SP. Consequently, after a trace t of SP, SUT may refuse all events refused by SP or accept some of them. Testing for $conf$ based on the refusals of SP would be, therefore, useless. What must be tested is that, after every trace t of SP, SUT cannot refuse all events in a set X of events such that $(t, X) \notin failures(SP)$. Such sets of events are called *acceptance sets* of SP after t.

Thus, tests for $conf$ are based on traces and acceptance sets. For a finite trace $s = \langle a_1, a_2, \ldots, a_n \rangle$ and a(n acceptance) set $X = \{x_1, \ldots, x_m\}$ of events, we define the *Circus* test process $T_F(s, X)$ as shown below.

$$\begin{aligned} T_F(s, X) = \ & inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow inc \ldots a_n \rightarrow fail \\ & \rightarrow (x_1 \rightarrow pass \rightarrow \mathbf{Stop} \,\Box \ldots \Box\, x_m \rightarrow pass \rightarrow \mathbf{Stop}) \end{aligned}$$

Example 3. *An example of such a test for Chrono, based on trace* $\langle tick \rangle$ *and on the acceptance set* $\{tick, time\}$ *is:*

$$\begin{aligned} T_F(\langle tick \rangle, \{tick, time\}) = \ & inc \rightarrow tick \rightarrow fail \\ & \rightarrow (tick \rightarrow pass \rightarrow \mathbf{Stop} \,\Box\, time \rightarrow pass \rightarrow \mathbf{Stop}) \end{aligned}$$

□

An exhaustive test set of a specification SP for $conf$ is made of all tests formed by considering all traces of SP, and all the acceptance sets after each of them:

$$\{\ T_F(t, X) \mid t \in traces(SP) \land (t, X) \notin failures(SP)\ \}$$

Actually $Exhaust_{conf}(\ SP)$ is defined as a subset of the set above where only minimal acceptance sets are considered, since as soon as some event in a set X is accepted after a trace, any set containing X is an acceptance set after this trace. For instance, the test in Example 3 is not in $Exhaust_{conf}(\texttt{Chrono})$, but the two tests below are:

$$T_F(\langle tick \rangle, \{tick\}) = inc \rightarrow tick \rightarrow fail \rightarrow (tick \rightarrow pass \rightarrow \mathbf{Stop})$$
$$T_F(\langle tick \rangle, \{time\}) = inc \rightarrow tick \rightarrow fail \rightarrow (time \rightarrow pass \rightarrow \mathbf{Stop})$$

The symbolic counterpart of $Exhaust_{conf}(\ SP)$ is $SExhaust_{conf}(SP)$, defined in [22].

$SExhaust_T(SP)$ and $SExhaust_{conf}(SP)$ provide bases for defining strategies for test selection as definitions of subsets of $Exhaust_T(SP)$ and $Exhaust_{conf}(SP)$ via uniformity or regularity hypotheses [23] and adequate instantiations [22], or coverage criteria of the specification [28]. This paper addresses fault-based selection techniques.

The *Circus* testing theory for traces refinement can be seen as a fault-based testing approach, because tests are constructed from (minimal) invalid traces. The fault considered in a particular test is very specific: a single forbidden continuation; the exhaustive test set considers all such possible faults. The theory for $conf$ testing considers other specific faults, namely refusals of the SUT that are not specified.

A practical question regards the use of more elaborated fault models. This is studied in the next section, and mutation operators are presented in Section 4.

3. Mutation testing in *Circus*

We assume that there is a specification (model), which is a *Circus* process that describes what the implementation should do. A *mutant* is also a *Circus* process, somehow related to the original specification; it represents a fault in that specification. In what follows, we may also refer to mutants as *faulty models*. In general, given a mutant defined as a *Circus* process, it can be used to generate tests to identify the fault it represents, that is, to "kill the mutant". We apply the approach in [36] by Aichernig at al. for mutation testing based on refinement, but consider the particular case of state-rich process algebraic models, and *Circus* in particular.

A mutant FM is *of interest* if there exists at least one test that can kill it. This is not the case if it is a refinement of the specification. This is the counterpart at the model level of the well known problem of equivalent mutants at the program level. In such a case, the mutant is not a faulty model and so not relevant.

In Section 3.1 we formalise mutation testing when traces refinement is the conformance relation of interest. Section 3.2 presents an example: a mutant and the tests they generate. Section 3.3 considers mutation when testing against the $conf$ conformance relation. Finally, in Section 3.4, we introduce a new kind of trace, closer to the text of the specification, to relate tests and mutation points in the specification.

3.1. Killing faulty models against traces refinement

In the context of traces refinement, we consider a faulty model FM to be of interest if it is not a traces refinement of the specification model SP. In this case, there is at least one trace of FM that is not a trace of SP; this is not the empty trace $\langle\rangle$, because $\langle\rangle$ is a trace of all processes. To detect the fault in FM, we can use a test $T_T(s, a)$ characterised by any of the minimal traces $s \frown \langle a \rangle$ of FM that are not traces of S. Formally, we define the set $FBTests_T^{SP}(FM)$ of fault-based tests characterised by FM with respect to a specification SP as shown below.

Definition 1.

$$FBTests_T^{SP}(FM) = \\ \{s : traces(SP);\ a : \Sigma \mid s \frown \langle a \rangle \in traces(FM) \setminus traces(SP) \bullet T_T(s, a)\}$$

This is the set of all tests $T_T(s, a)$, formed from traces s of SP and events a from Σ such that $s \frown \langle a \rangle$ is a trace of FM, but not of SP. A mutant FM is killed by any test from $FBTests_T^{SP}(FM)$. As a direct consequence of its definition, we have that $FBTests_T^{SP}(FM)$ is a subset of $Exhaust_T(SP)$, since it contains tests $T_T(s, a)$, where $s \in traces(SP)$ and $s \frown \langle a \rangle \notin traces(SP)$.

Before generating tests based on a mutant FM, a first step is the confirmation that it is not a traces refinement of the model SP. For that, it is of value to use a refinement model checker, like FDR [43] for CSP, for example, to check $SP \sqsubseteq_T FM$. If this does not hold, FDR provides a counterexample: a minimal trace of FM that is not a trace of SP. As said above, this identifies a test to detect the fault specified in FM.

$$
\begin{array}{l}
\textbf{process}\ \mathit{MutatedChrono} \mathrel{\widehat{=}} \\
... \\
\quad \mathit{Run} \mathrel{\widehat{=}}\ \ (\mathit{tick} \rightarrow \mathit{IncSec};\ \ (\neg\ (\mathit{sec} = 0)\ \&\ \mathit{IncMin}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box((\mathit{sec} \neq 0)\ \&\ \mathit{Skip}))) \\
\qquad\qquad\quad \Box(\mathit{time} \rightarrow \mathit{out}\,!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}) \\
\quad \bullet\ (\mathit{AInit};\ (\mu\ X \bullet (\mathit{Run};\ X))) \\
\quad ...
\end{array}
$$

Figure 4: a mutated chronometer

3.2. A first mutant of Chrono and some tests that kill it

Figure 2 presents the mutant $MutatedChrono$ of the $Chrono$ process, which is obtained by the introduction of the negation ($\neg$) operator in the first guard of the action Run. The following change in behavior arises from this mutation: like $Chrono$, $MutatedChrono$ starts with the $AInit$ operation that initialises min and sec to 0 and, after a $tick$, $IncSec$ increases the value of sec to 1; afterwards, however, the mutant behaves nondeterministically, either like $Chrono$, that is, executing **Skip** and then Run again, or performing $IncMin$ and then Run again, like $Chrono$ in the case $sec = 0$. This erroneously leads to a state where $min = 1$ and $sec = 1$. This mutated state can be later observed via an output on the channel out following a $time$ event.

Thus $\langle tick, time, out!(1,1)\rangle$ is a trace of $MutatedChrono$, but not of $Chrono$, for which the only accepted event after $\langle tick, time\rangle$ is $out!(0,1)$. As seen above such traces define tests that kill the mutant. An example is the test below:

$$inc \rightarrow tick \rightarrow inc \rightarrow time \rightarrow pass \rightarrow out!(1,1) \rightarrow fail \rightarrow \textbf{Stop}$$

It is a member of $FBTests_T^{Chrono}(MutatedChrono)$.

Actually, this test *may* kill the mutant since it is nondeterministic due to the overlap of the guards in the choice operator (for the distinction between "may kill" and "must kill", see [44]). The test kills the mutant under the complete testing assumption.

3.3. Killing faulty models against the conf conformance relation

Given a mutant FM, it is of interest if $\neg\ (FM\ conf\ SP)$. In this case, there is at least one common trace s of SP and FM for which $\neg\ (Ref(FM, s) \subseteq Ref(SP, s))$. Therefore, according to the definition of $Ref(P, s)$, there is at least one set of events X such that $(s, X) \in failures(FM)$, but $(s, X) \notin failures(SP)$.

The detection of the fault specified in FM with respect to $conf$ is based on tests $T_F(s, X)$ characterized by $(s, X) \in failures(FM)$, where $(s, X) \notin failures(SP)$, and s is a trace s of SP and FM. The full set $FBTests_F^{SP}(FM)$ of fault-based tests for $conf$ characterized by FM with respect to SP is defined as follows.

Definition 2.

$$FBTests_F^{SP}(FM) = \\ \{s : traces(SP) \cap traces(FM);\ X : \mathbb{P}\Sigma \mid \\ (s, X) \in failures(FM) \setminus failures(SP) \bullet T_F(s, X)\}$$

As a direct consequence of the above definition, we have that $FBTests_F^{SP}(FM)$ is a subset of $Exhaust_{conf}(SP)$, since it contains tests $T_F(s, X)$, where $s \in traces(SP)$ and $(s, X) \notin failures(SP)$. In words, X is an acceptance set of SP after s.

Theorem 1. *For every mutant FM of a specification SP, the following statements are equivalent:*

1. *FM is of interest; and*
2. $FBTests_T^{SP}(FM) \neq \emptyset \vee FBTests_F^{SP}(FM) \neq \emptyset$.

Proof. If FM is of interest, it is not a refinement of SP, that is, either it is not a traces refinement of SP, or it does not satisfy $FM\ conf\ SP$.

In the first case, as shown in [22], there exists some $T \in Exhaust_T(SP)$ such that its execution against FM yields a $fail$ verdict. By definition of $Exhaust_T(SP)$, there are s and a, such that $T = T_T(s, a)$ and $s \in traces(SP) \wedge s^\frown\langle a\rangle \notin traces(SP)$. From the definition of $T_T(s, a)$, since the $fail$ event is reached, $s^\frown\langle a\rangle$ is a trace of FM. Thus, from Definition 1, T belongs to $FBTests_T^{SP}(FM)$, and so $FBTests_T^{SP}(FM) \neq \emptyset$.

With a similar argument, the second case implies $FBTests_F^{SP}(FM) \neq \emptyset$.

Conversely, if $FBTests_T^{SP}(FM) \neq \emptyset$, there exists some $T = T_T(s, a)$, where $s \in traces(SP)$ and $s^\frown\langle a\rangle \in traces(FM) \setminus traces(SP)$. T belongs to $Exhaust_T(SP)$ and by construction yields a $fail$ verdict when executed against FM. Therefore, from the exhaustivity result of [22], FM is not a traces refinement of SP.

The proof that $FBTests_F^{SP}(FM) \neq \emptyset$ implies that $\neg\ FM\ conf\ SP$ is similar. □

In Theorem 1, we formalise the previously introduced notion of mutants of interest: a mutant is of interest if its fault can be exposed by, at least, a test.

Example 4. *Coming back to $MutatedChrono$, another change of behavior is that it introduces a deadlock. When the mutated external choice is reached in a state where $sec = 0$, because all the possible choices are guarded by the negation of this condition, there is a deadlock. Due to the equation $sec' = (sec + 1) mod\ 60$ in the $IncSec$ schema, it occurs after sixty $tick$ events. After such a trace, $Chrono$ must accept one more $tick$ event, or one $time$ event, but the mutated specification refuses both. This leads to the following tests, each of them killing the mutant under the complete testing assumption.*

$$(inc \rightarrow tick)^{60} \rightarrow fail \rightarrow tick \rightarrow pass \rightarrow \textbf{Stop}$$
$$(inc \rightarrow tick)^{60} \rightarrow fail \rightarrow time \rightarrow pass \rightarrow \textbf{Stop}$$

We use $(inc \rightarrow tick)^{60}$ to denote a prefixing action where the events inc and $tick$ are offered in alternation, starting with inc, 60 times. □

When generating tests based on given a mutant FM, a first step is the confirmation that it is a traces refinement, but not a failures refinement of SP. If we can use a refinement model checker, we can check $SP \sqsubseteq_T FM$, and then use the counterexample for $SP \sqsubseteq_F FM$. In the case of FDR, for example, the counterexample is a minimal set of acceptances of FM that is not a set of acceptances of SP. Alternatively, it gives a set of refusals of FM that is not a set of refusals of SP. It is this set of refusals that can be directly used to define a test to detect the fault specified in FM.

In our framework, a set of mutants M of SP defines a set

$$T_T^M \mathrel{\widehat=} \{M \bullet FBTests_T^{SP}(M)\}$$

of subsets of $Exhaust_T(SP)$ and a set

$$T_F^M \mathrel{\widehat=} \{M \bullet FBTests_F^{SP}(FM)\}$$

of subsets of $Exhaust_{conf}(SP)$. Therefore, M is the basis of a test selection method in the sense of [22]. Besides, given a test suite T, it is adequate for M if, for each $m \in M$, $T \cap FBTests_T^{SP}(m) \neq \emptyset$ or $T \cap FBTests_F^{SP}(m) \neq \emptyset$. From Lemma 1, we know that for a set M of mutants of interest, there is always an adequate test suite.

3.4. Specification traces and mutation points

Mutations are related to the text of a specification. Traces and even constrained symbolic traces, however, are not related to the text of the specification. They record a possible history of interactions, and it may well be the case that, in some specific situations, we can relate interactions to events and communications in the text of specification. On the other hand, but there is no record of guards and data operations that may have been evaluated or executed in the path to that interaction.

As an example, we consider the constrained symbolic trace $cst3$ in Figure 3. Since there is only one communication via out in the text of $Chrono$, in this special case, we can relate $out!\alpha_0!\alpha_1$ to the anti-penultimate line of its definition. On the other hand, $cst3$ has no record of $AInit$ and $IncSec$, and of the guards $sec = 0$ and $sec \neq 0$, which are considered in the path to that interaction and may be the object of a mutation.

Therefore, we use *specification traces*, as defined in [45, 28], to build the tests aimed at killing a mutant. In [28], specification traces are used to consider data-flow coverage, which is also based on the text of a specification. While cstraces are useful for trace selection based on constraints on the traces, they do not support selection based on the text of the specification, as we explain in the sequel.

In specification traces, labels are pieces of the specification: guards (predicates), communications, data operations (schemas) or simple *Circus* actions. In case there are repetitions of identical text pieces in the *Circus* specification, different occurrences are distinguished in the labels using textual tags. The syntactic category of $Label$s is defined in Figure 5; the sets $Pred$, Exp, $CName$, $VName$, and $Schema$ are those of the *Circus* predicates, expressions, channel and variable names, and Z schemas [46].

In Figure 6, we present two rules of the transition system that characterises specification traces. These transition rules correspond to those for the operational semantics shown in Figure 2. We note that, the same notion of configuration is used and the transitions are the same, except for the labels. For instance, for an input, the label $d?w_0$

$$
\begin{array}{lcl}
\mathit{Label} & ::= & \mathit{Pred} \mid \mathit{Comm} \mid \mathit{LAct} \\
\mathit{Comm} & ::= & \epsilon \mid \mathit{CName} \mid \mathit{CName}!\mathit{Exp} \mid \mathit{CName}?\mathit{VName} \\
 & & \mid \mathit{CName}?\mathit{VName} : \mathit{Pred} \\
\mathit{LAct} & ::= & \mathit{VName}^* : [\mathit{Pred}, \mathit{Pred}] \mid \mathit{Schema} \mid \mathit{VName} := \mathit{Exp} \\
 & & \mid \mathtt{var}\ \mathit{VName} : \mathit{Exp} \mid \mathtt{var}\ \mathit{VName} := \mathit{Exp} \mid \mathtt{end}\ \mathit{VName}
\end{array}
$$

Figure 5: Syntax of specification labels.

$$
\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x} : \mathtt{T} \rightarrow \mathtt{A}) \overset{\mathtt{d?x}}{\Longrightarrow} (\mathtt{c} \wedge \mathtt{w_0} \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\ x} := \mathtt{w_0} \models \mathtt{let\ x} \bullet \mathtt{A})} \tag{1}
$$

$$
\frac{c \wedge (s;\ g)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{g} \,\&\, \mathtt{A}) \overset{\mathtt{g}}{\Longrightarrow}_P (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{g}) \mid \mathtt{s} \models \mathtt{A})} \tag{2}
$$

Figure 6: Transition rules that define specification traces

in the operational semantics uses a symbolic variable w_0, while in the specification traces it refers to the variable x used in the specification. Moreover, the transition for a guarded action $g \,\&\, A$ is no longer unlabelled, but records the guard g.

In Figures 7 and 8, we list some specification traces for the processes Chrono and $\mathit{MutatedChrono}$. The use of text pieces from the specifications in labels allows the identification and selection of traces that reach textual constructs affected by mutation.

Remark. In [28], a subset of specification traces, *sptraces*, is considered for the definition of tests satisfying data-flow coverage criteria, namely, the set of specification traces where the last event is an observable event. Some of the specification traces given in Figures 7 and 8 are not sptraces. It is the case of $\mathit{spect}1$ and $\mathit{spect}'1$, and of all the traces in $\mathit{Setspect}'5$. The specification traces in this last set, for example, are of great interest since they lead to a deadlock that is not in the original specification, and thus provide bases for obtaining tests of $\mathit{FBTests}_F^{\mathit{Chrono}}(\mathit{MutatedChrono})$. □

Converting a specification trace to a cstrace requires the definition of an operational semantics for labels. Figure 9 presents its transition rules for input and guard labels. We refer to Figure 2 for the corresponding rules of the operational semantics. Like in the operational semantics, the configuration is a triple, but here, instead of a process or action, there is a label associated with a constraint and a state assignment. Labels with no guard, but with an input or output communication, are handled in the same way as input and output prefixes in the operational semantics. When there is a label $(\mathtt{g}, \mathtt{e}, \mathtt{A})$, with a guard that may be different from `True`, if the guard holds in the current state then there is a transition to a label $(\mathtt{e}, \mathtt{A})$ with guard `True`, or no guard, for short. The

$$
\begin{aligned}
&spect1 : \langle \texttt{AInit}, tick, IncSec, (sec \neq 0)\rangle \\
&spect2 : \langle \texttt{AInit}, time, out!min!sec\rangle \\
&spect3 : \langle \texttt{AInit}, tick, IncSec, (sec \neq 0), time, out!min!sec\rangle \\
&spect4 : \langle \texttt{AInit}\rangle \frown \langle tick, IncSec, (sec \neq 0)\rangle^{59} \frown \langle tick, IncSec, (sec = 0), IncMin\rangle \\
&spect5 : \langle \texttt{AInit}\rangle \frown \langle tick, IncSec, (sec \neq 0)\rangle^{59} \frown \\
&\qquad\qquad \langle tick, IncSec, (sec = 0), IncMin, time, out!min!sec\rangle
\end{aligned}
$$

Figure 7: Some specification traces for $Chrono$

$$
\begin{aligned}
&spect'1 : \langle AInit, tick, IncSec, \neg\ (sec = 0), IncMin\rangle \\
&spect'2 : \langle AInit, time, out!min!sec\rangle \\
&spect'3 : \langle AInit, tick, IncSec, (sec \neq 0), time, out!min!sec\rangle \\
&spect'4 : \langle AInit, tick, IncSec, \neg\ (sec = 0), IncMin, time, out!min!sec\rangle \\
&Setspect'5 : \langle AInit, tick, IncSec\rangle \frown \\
&\qquad\qquad \big(\langle \neg\ (sec = 0), IncMin, tick, IncSec\rangle \mid \langle (sec \neq 0), tick, IncSec\rangle\big)^{59} \\
&\qquad\qquad [deadlock]
\end{aligned}
$$

Figure 8: Some specification traces for $MutatedChrono$

transition is unlabelled, like in the operational semantics.

The similarity between the operational semantics of labels and of *Circus* is not surprising. Conversion of specification traces (of labels) to a cstrace recovers the operational semantics of the *Circus* texts captured in the specification traces.

In Figures 3 and 10 we give the constrained symbolic traces converted from those specification traces listed in Figures 7 and 8. The trace $cst'4$ is that used in Section 3.2 as a basis for the first test that kills $MutatedChrono$. The trace $cst'5$ is the unique translation of all the traces in $Setspect'5$; it is the basis of the two other killer tests given in Section 3.2. The conversion procedure of sptraces into cstraces is given in [28] and trivially generalises to specification traces. As seen above, several specification traces may correspond to the same cstrace. This follows from the fact that cstraces record only observable symbolic events, while specification traces record internal events and may distinguish different ways of enchaining the same observable events.

To summarise, in considering mutation testing in *Circus* we use three kinds of traces: specification traces, cstraces, and standard traces. We perform the selection among specification traces and then generate from those some killer tests belonging to $FBTests_T^{SP}(FM)$ and $FBTests_F^{SP}(FM)$, which are defined as sets of concrete tests based on standard traces. Since the operational semantics defines the traces of a speci-

$$\frac{c \wedge (s;\ g)}{(\mathtt{c} \mid \mathtt{s} \models (\mathtt{g}, \mathtt{e}, \mathtt{A})) \xrightarrow{\epsilon} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{g}) \mid \mathtt{s} \models (\mathtt{e}, \mathtt{A}))}$$

$$\frac{c \wedge T \neq \varnothing}{(\mathtt{c} \mid \mathtt{s} \models (\mathtt{d?x} : \mathtt{T}, \mathtt{A})) \xrightarrow{\mathtt{d?w_0}} (\mathtt{c} \wedge \mathtt{w_0} \in \mathtt{T}) \mid \mathtt{s};\ \mathtt{var\ x} := \mathtt{w_0} \models \mathtt{let\ x} \bullet \mathtt{A})}$$

Figure 9: Operational semantics of labels

$$cst'1 : (\langle tick \rangle, true)$$
$$cst'2 : (\langle time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 0 \wedge \alpha_1 = 0)$$
$$cst'3 : (\langle tick, time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 0 \wedge \alpha_1 = 1)$$
$$cst'4 : (\langle tick, time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 1 \wedge \alpha_1 = 1)$$
$$cst'5 : (\langle tick \rangle^{60}, true) \qquad [deadlock]$$

Figure 10: Constrained symbolic traces for $MutatedChrono$

fication as cstraces, there is an intermediate form of the tests based on cstraces, that is, symbolic killer tests, that need to be adequately instantiated into some concrete tests.

We must take into account that a specification trace corresponds to one cstrace, but that several specification traces may correspond to the same cstrace, as seen above, and that a specification trace of FM that is not a specification trace of SP may lead to a cstrace that is the translation of another specification trace of SP. Given a mutant FM of SP, we define a notion of relevant specification trace of FM.

Definition 3. Given a specification SP and a faulty model FM obtained by mutation of SP, a specification trace of FM is *relevant* if:

- it is not a specification trace of SP and its conversion into a cstrace is not a cstrace of SP, or
- one of the instantiations of its conversion into a cstrace is part of some failure of FM that is not a failure of SP.

The first case is the starting point for obtaining some tests in $FBTests_T^{SP}(FM)$ and the second case for some tests in $FBTests_F^{SP}(FM)$.

To determine whether a specification trace of FM is relevant, we need to compare its cstrace to those of SP. We note that in the *Circus* testing theory, we use unique (up to predicate equivalence) symbolic representatives of these traces using a fixed ordered alphabet of symbolic variables. That allows a uniform account of these traces, and simplifies the management of names by avoiding renaming complications.

If there are no relevant specification traces of FM, it means that there exist no test for killing FM, that is, the effect of the mutation is not observable, neither for traces, nor for failures. FM is a refinement of SP and is not a mutant of interest.

The existence of non-relevant mutated specification traces is due to the possibility of specifying systems in an abstract way, with hidden operations on some state: the consequences of a mutation of such operations may not be observable. This phenomenon is not specific to *Circus* and similar issues are likely to occur in model-based mutation testing as soon as there is a significant abstraction gap between the model and the SUT.

It is very likely that some mutations affecting the hidden state can be identified as prone to producing non-relevant specification traces, and then avoided in a particular test-selection heuristic. These points are the subject of future work.

In conclusion, given a mutation of a *Circus* specification, the principle of the test-generation process that we propose is as follows:

1. Select a specification trace $spect$ of FM that reaches the mutation point;
2. Convert it into its corresponding cstrace cst;
3. Check whether cst is a cstrace of SP;
4. If not, instantiate cst into a concrete trace and build the corresponding tests of $FBTests_T^{SP}(FM)$ as defined in Section 3.1;
5. Check whether some instantiations of cst leads to failures that are not failures of SP and build the corresponding tests of $FBTests_F^{SP}(FM)$, as defined in Section 3.3.

We discuss in Section 5 some tools to support this process. First, in the next section, we present operators to produce mutants.

4. Mutation operators

A mutation operator Op is a function that generates a set of mutants for a given specification SP. In each mutant in $Op(SP)$, one fault is inserted. Such an operator only yields well-typed, syntactically correct, mutants. A mutation operator is *valid* for a specification, if it generates at least one mutant of interest.

In this section, we present a list of mutation operators for *Circus*. Many of the mutation operators of CSP presented in [31] by Srivatanakul et al. are directly applicable to *Circus*. These are discussed in Section 4.1. For the data and state aspects of *Circus*, we can benefit from some mutation operators based on fault classes for predicates, like those presented by Kuhn in [32] and enriched by Black et al. in [33]. These are discussed in Section 4.2. We need to take into consideration, however, the specific features and particularities of our target language, *Circus*.

4.1. Modification of behavioural operators

Three classes of mutation operators are suggested for CSP in [31]: process definition, expression, and parameter modification operators. Each operator from the first two classes is considered in this section, and we introduce some variations better suited to *Circus* models. When adequate, we refer to the rules of the *Circus* operational semantics that formalise the affected behaviors.

Parameters are not as important in *Circus* as they are in CSP. Since *Circus* processes can have a state, parametrisation is typically used to define generic processes. In this

Class		Abbreviation	Operator	Source	OPS Rules
Behavioural Operators	Events	ped	Event Drop	CSP	9, 30
		per	Event Replacement	CSP	9, 30
		pes	Event Swap	CSP	9, 30
		pei	Event Insert	CSP	9, 30
	Operators	pco	Choice Operator	CSP	18 - 24
		ppoParSeq	Parallel Composition	CSP*	14,15,25-30
		ppoSeqPar	Sequential Composition	CSP*	25 - 30
		ppoSeqInt	Interleave	CSP*	25 - 30
		ppoParInt	Interleave	CSP*	25 - 30
		ppoNameSet	Parallel State Writing	*Circus*	25 - 30
	Communic.	pmr	Message Replacement	CSP	10 - 12
		pcr	Channel Replacement	CSP	10 - 12
		pci	Communication Insert	CSP	10 - 12
		pce	Communication Elimination	CSP	10 - 12
		pcs	Communication Swap	CSP	10 - 12
	Name/Hide	pprAction	Action Name	*Circus*	
		pprSchema	Schema Name	*Circus*	
		pprHide	Hide Events	*Circus*	31 - 33
		pprUnhide	Unhide Events	*Circus*	31 - 33
Expression Operators	Logical	eni	Negation Insert	CSP	
		eniGuard	Guard Negation	CSP*	17
		elr	Operator Replacement	CSP	
		eld	Operand Replacement	CSP	
	Arithmetical	ear	Operator Replacement	CSP	
		eur	Unary Insertion	CSP	
		eak	Add k to Operand	CSP	
		esk	Sub k from Operand	CSP	
		ead	Operand Replacement	CSP	
	Rel.	err	Operator Replacement	CSP	

Table 1: Mutation operators for *Circus*

case, parameters play the role of global constants in the scope of the process definition. We, therefore, do no consider parameter operators here.

Following the terminology in [31], synchronization events, that is, communications without value passing, are called simply "events". When values are passed, we refer to the events as "communications". Table 1 lists the operators presented in this section, identifying their classification, abbreviation, name, source and, when available in Appendix A, the associated operational semantics rules. The source column displays CSP for operators originally proposed for CSP, CSP* for operators adapted from CSP to match *Circus* constructs, and *Circus* for operators designed specifically for *Circus*.

We describe the operators as functions from texts (of actions) to texts, and identify conditions in which they can or cannot be applied. A mutated process is obtained by applying one of these functions to one of its actions.

$$
\begin{array}{l}
\textbf{process}\ \mathit{ChronoEDropTime} \mathrel{\widehat=} \\
\ldots \\
\quad \mathit{Run} \mathrel{\widehat=} \quad (\mathit{tick} \rightarrow \mathit{IncSec};\quad ((\mathit{sec} = 0)\ \&\ \mathit{IncMin}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box((\mathit{sec} \neq 0)\ \&\ \mathit{Skip}))) \\
\qquad\qquad\quad \Box(\mathit{out}\,!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}) \\
\quad \bullet\ (\mathit{AInit};\ (\mu\ X \bullet (\mathit{Run};\ X))) \\
\quad \ldots
\end{array}
$$

Figure 11: Mutation of $Chrono$ after dropping $time$

In the sequel, we present mutations of events and communications, choice and concurrency operators, name references, and hiding.

4.1.1. Mutations of events

Event Drop. (ped) removes one (arbitrary) occurrence of a prefixing event. Such mutation is likely to produce processes that are not a refinement of the original specification in two ways: the removal of observable events from some traces and the possibility of deadlock introduction in parallel compositions. This can be checked in the rules (A.2) and (A.6) of the *Circus* operational semantics, reproduced in Appendix A, where output and parallelism are described. The definition of the ped operator is as follows.

$$ped(\mathtt{A}[\mathtt{e}\rightarrow]) = \mathtt{A}[\,]$$

Remark. We use $\mathtt{A}[\mathtt{t}_1]$ to indicate that in the text of action A there is an occurrence of term $\mathtt{t}_1$. There may be several such occurrences and we assume that we can distinguish them (via their position in the text, for example). $\mathtt{A}[\mathtt{t}_1]$ refers to a particular occurrence. A subsequent reference to $\mathtt{A}[\mathtt{t}_2]$ denotes the action obtained by replacing the occurrence of $\mathtt{t}_1$ originally singled out by $\mathtt{t}_2$. In particular, $\mathtt{A}[\,]$ as used above denotes the text of A with this occurrence replaced by an empty term. □

In using the above operator and others to follow, the terms singled out in the action parameter, like $\mathtt{e}\rightarrow$ in $\mathtt{A}[\mathtt{e}\rightarrow]$ above, for example, need to be chosen.

Example 5. *Figures 11 and 12 show the mutants of the Chrono process obtained via the mutations:* $ped(\mathtt{Chrono}[\mathtt{time}\rightarrow])$ *and* $ped(\mathtt{Chrono}[\mathtt{tick}\rightarrow])$*, which remove the occurrences of the synchronization events* $time$ *and* $tick$ *from the* Run *action. Among the traces of* $ChronoEDropTime$ *that are not traces of Chrono, there are* $\langle \mathtt{out}!(\mathtt{0},\mathtt{0})\rangle$ *and* $\langle \mathtt{tick}, \mathtt{out}!(\mathtt{0},\mathtt{1})\rangle$*, which lead to the tests below:*

$$\mathit{pass} \rightarrow \mathit{out}!(0,0) \rightarrow \mathit{fail} \rightarrow \textbf{Stop}$$
$$\mathit{inc} \rightarrow \mathit{tick} \rightarrow \mathit{pass} \rightarrow \mathit{out}!(0,1) \rightarrow \mathit{fail} \rightarrow \textbf{Stop}$$

They are members of $FBTests_T^{Chrono}(ChronoEDropTime)$.

$$
\begin{array}{l}
\textbf{process}\ \mathit{ChronoEDropTick} \mathrel{\widehat{=}} \\
\ldots \\
\quad \mathit{Run} \mathrel{\widehat{=}}\ (\mathit{IncSec};\ ((\mathit{sec} = 0)\ \&\ \mathit{IncMin}) \\
\qquad\qquad\qquad\qquad \Box((\mathit{sec} \neq 0)\ \&\ \mathit{Skip}))) \\
\qquad\qquad \Box(\mathit{time} \rightarrow \mathit{out}\,!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}) \\
\quad \bullet\ (\mathit{AInit};\ (\mu\ X \bullet (\mathit{Run};\ X))) \\
\quad \ldots
\end{array}
$$

Figure 12: Mutation of $Chrono$ after dropping $tick$

$ChronoEDropTick$ is an example of a divergent mutant, since Run may recurse indefinitely in an endless series of internal actions $IncSec$ and $IncMin$. without ever communicating with the environment. Thus, it is discarded.

□

Event Replacement. (per) replaces an event by another one within the current local scope (action) or the global scope (process). It affects traces in similar ways as the ped operator, as described by the same rules of the operational semantics.

$$per(\mathtt{A[e\rightarrow]}, \mathtt{f}) = \mathtt{A[f\rightarrow]}$$

Event Swap. (pes) swaps two consecutive synchronization events in an action definition. If we apply it only to distinct events, this operator is prone to yielding mutants of interest, although this cannot be assured. For example, swapping `a` and `b` in $\mathtt{a} \rightarrow \mathtt{b} \rightarrow \mathtt{STOP} \,\Box\, \mathtt{b} \rightarrow \mathtt{a} \rightarrow \mathtt{STOP}$ leads to a traces refinement. The behavior is defined by the same rules (A.2) and (A.6) of the *Circus* operational semantics.

$$pes(\mathtt{A[e} \rightarrow \mathtt{f\rightarrow]}) = \mathtt{A[f} \rightarrow \mathtt{e\rightarrow]}\ where\ \mathtt{e} \neq \mathtt{f}$$

Event Insert. (pei) inserts an event by duplication. The mutated action may not be of interest if the duplication occurs inside a loop.

$$pei(\mathtt{A[e\rightarrow]}) = \mathtt{A[e} \rightarrow \mathtt{e\rightarrow]}$$

This concludes our list of event mutation operators.

4.1.2. Mutations of choice and concurrency operators

The following operators target choice and concurrency operators.

Choice Operator. (pco) replaces the external choice by the internal choice operator. From rules (A.4) and (A.8) to (A.11) of the operational semantics, we observe that such mutation may introduce deadlocks. The traces of the original and mutated actions are the same, so we always have a traces refinement. Therefore, the only error that may be introduced by such a mutation is a forbidden deadlock.

$$pco(\mathtt{A[}\Box\mathtt{]}) = \mathtt{A[}\sqcap\mathtt{]}$$

We note that replacing an internal with an external choice is not of interest, since $A \sqcap B$

is refined by $A \,\Box\, B$, whether we consider traces or failures refinement.

Parallelism and sequence. (*ppo*) In [31], this operator can be used to replace an interleaving, a parallel or a sequential composition with each other. For *Circus*, we need to consider manipulation of state values in concurrent actions. We, therefore, distinguish the possible mutations for each type of composition.

Parallel composition of actions requires the explicit specification of which variables are available for writing by each concurrent action. For instance, in the *Circus* parallel composition $\mathtt{A} \,[\![\, \mathtt{NS_a} \mid \mathtt{CS} \mid \mathtt{NS_b} \,]\!]\, \mathtt{B}$ values written by A into the state variables are only kept for the variables with names in the name set NS_a. The same holds for the action B and the name set NS_b. These sets must be disjoint. Any change of value not matching a name in the respective specified set is cancelled after the parallel composition ends (see rules (A.5) to (A.7) of the operational semantics). There are several interesting possibilities for injecting faults in parallel composition operators. For instance, to empty one or both name sets in a parallel composition is likely to yield unexpected state configurations and is defined by the following operation.

$$ppoNameSet(\mathtt{A[NS]}) = \mathtt{A[\{\}]}$$

NS refers to a name set in a parallel composition. Removing or inserting arbitrary elements in NS and other variations of this operator might also be of interest, provided the disjointness of the sets on both sides of the composition is preserved.

We define some specific operators for mutating the nature of the compositions. Such mutations are likely to cause substantial observable changes, although there is no guarantee that the result is a mutant of interest. Specific operators for changing parallel to sequential and the inverse are defined as follows.

$$ppoParSeq(\mathtt{A[B} \,[\![\, \mathtt{NS_b} \mid \mathtt{CS} \mid \mathtt{NS_c} \,]\!]\, \mathtt{C]}) = \mathtt{A[B;\ C]}$$
$$ppoSeqPar(\mathtt{A[;\ ]}, \mathtt{CS}) = \mathtt{A}[\![\mathtt{CS}]\!]$$

The new sequential composition introduced by $ppoParSeq$ ignores the extra parameters of the original parallel composition, that is, $\mathtt{NS_a}$, CS, and $\mathtt{NS_b}$, and the new parallel composition introduced by $ppoSeqPar$ is designed to synchronize on all channels in a given set CS and has no write access to any state values. We also define operators for mutating any composition into an interleaving.

$$ppoParInt(\mathtt{A}[\![\mathtt{CS}]\!]) = \mathtt{A}[|||]$$
$$ppoSeqInt(\mathtt{A[;\ ]}) = \mathtt{A}[|||]$$

Other forms of parallelism may be considered as well, but we do not pursue these here. The needed considerations in all cases are similar to those above.

4.1.3. Mutations of communications

The following mutation operators target communications, whose behaviours are described in rules (A.2) and (A.3). For the sake of conciseness, we do not discuss communications with multiple inputs or outputs.

The removal of an input communication can potentially introduce a syntax error, because it implicitly declares a new variable. Except when the input variable is never

used, we may end up with references to variables that are not declared. For mutation operators that change or remove input communications, we, therefore, introduce a declaration of the replaced or removed variable. A *Circus* variable declaration introduces the variable in scope initialised with an arbitrary value from the variable domain.

Message Replacement. (pmr) changes the name of a variable used in a communication to another name of a variable of the same type, selected from the current scope. As explained above, missing variable names due to the introduced mutation are redeclared.

$pmr(\mathtt{A[c!e[x]\rightarrow], y}) = \mathtt{A[c!e[y]\rightarrow]}$
$pmr(\mathtt{A[c?x\rightarrow], y}) = \mathtt{A[var\ x : T \bullet c?y\rightarrow]}$

T is the type of channel c (and, therefore, of the input variable x).

Channel Replacement. (pcr) changes the name of a communication channel to another channel name of the same type.

$pcr(\mathtt{A[c!e\rightarrow], d}) = \mathtt{A[d!e\rightarrow]}$
$pcr(\mathtt{A[c?x\rightarrow], d}) = \mathtt{A[d?x\rightarrow]}$

Communication Insert. (pci) is similar to the event insert (pei) operator, but applied to a communication instead of a synchronization event. It inserts a new communication by duplicating an existing one.

$pci(\mathtt{A[c!e\rightarrow]}) = \mathtt{A[c!e \rightarrow c!e\rightarrow]}$
$pci(\mathtt{A[c?x\rightarrow]}) = \mathtt{A[c?x \rightarrow c?x\rightarrow]}$

Communication Elimination. (pce) removes one arbitrary input or output communication from an action definition. As for the message replacement (pmr) operator, if the eliminated communication is an input, a variable declaration must be introduced to declare the input variable and avoid syntactic errors.

$pce(\mathtt{A[c?x\rightarrow]}) = \mathtt{A[var\ x : T \bullet]}$
$pce(\mathtt{A[c!e\rightarrow]}) = \mathtt{A[\,]}$

If the mutated action terminates or deadlocks, it has maximal traces. If the eliminated communication contributes only to the last events of maximal traces, its elimination leads to an action that is a traces refinement of the original action.

Communication Swap. (pcs) swaps two consecutive communication events, similarly to the event swap (pes) operator.

$pcs(\mathtt{A[c1!e1 \rightarrow c2!e2\rightarrow]}) = \mathtt{A[c2!e2 \rightarrow c1!e1\rightarrow]}$
$pcs(\mathtt{A[c1!e \rightarrow c2?x\rightarrow]}) = \mathtt{A[c2?x \rightarrow c1!e\rightarrow]},\ \textit{provided}\ \mathtt{x}\ \textit{is not free in}\ \mathtt{e}$
$pcs(\mathtt{A[c1?x \rightarrow c2?y\rightarrow]}) = \mathtt{A[c2?y \rightarrow c1?x\rightarrow]}$
$pcs(\mathtt{A[c1?x \rightarrow c2!e\rightarrow]}) = \mathtt{A[var\ x : T \bullet c2!e \rightarrow c1?x\rightarrow]}$

When swapping $c1$ and $c2$ where $c1$ is used in an input that declares a variable that may be used in an expression communicated by $c2$, this change leads to a syntactic error. To avoid that, a declaration of the input variable is introduced.

4.1.4. Mutations of name references and hiding

Name Replacement. (ppr) substitutes name references in the right hand side of an action definition by other process names in scope, including $STOP$ and $SKIP$. For *Circus*, we expand this operator to include the manipulation of schema names.

$$pprAction(\mathtt{A[A1]}, \mathtt{A2}) = \mathtt{A[A2]}$$
$$pprSchema(\mathtt{A[S1]}, \mathtt{S2}) = \mathtt{A[S2]}$$

We also introduce two new operators for hiding and unhiding events and communications. The impact of such mutations on observable behavior is similar to event and communication insertion or removal. The channel set for the hiding is an arbitrary subset of the channels in scope. The operational semantics rules describing the behavior of the hiding operator are (A.12) and (A.13).

$$pprHide(\mathtt{A1[A2]}, \mathtt{CS}) = \mathtt{A1[A2} \setminus \mathtt{CS]}$$
$$pprUnhide(\mathtt{A1[A2} \setminus \mathtt{CS]}) = \mathtt{A1[A2]}$$

Similar operators for hiding are also useful at the action level.

4.2. Expression modification operators

In *Circus* specifications, some logical and data aspects are modelled along with the definitions of interactions via events. For instance, models typically include guard predicates, variable definitions and assignments, and also data operations defined using Z schemas. We consider all these forms of data modelling in the design of mutation operators that capture data and logic faults.

Some fault classes for predicates occurring in software specifications have been proposed by Kuhn in [32]. They seem pertinent if we consider plausible modelling mistakes in *Circus*: variable reference, variable negation, expression negation, associative shifting, operator reference, and missing expressions. We also consider the expression operators introduced for CSP in [31], since they cover most of the mentioned fault classes. Finally, to complement the fault-classes coverage, we also introduce some syntactical operators for *Circus* expressions inspired by the work of Black et al. [33].

Negation Insertion. (eni) inserts the logical negation operator ($\neg$) before a boolean variable. This mutation is specially interesting when it affects guards, so we have a more specific version of this operator targeting the negation of guards only.

$$eni(\mathtt{A[e]}) = \mathtt{A[}\neg\ \mathtt{e]}$$
$$eniGuard(\mathtt{A[g}\&\mathtt{]}) = \mathtt{A[}\neg\ \mathtt{g}\&\mathtt{]}$$

where e is a boolean expression in any *Circus* context, an action or a schema. For example, $MutatedChrono$ is obtained from $Chrono$ via $eniGuard(\mathtt{Chrono[(sec = 0)}\&\mathtt{]})$.

Logical Operator. (elr) exchanges between the logical "and" ($\wedge$) and "or" ($\vee$) operators. Other logical connectors might be considered, although the most used and more subject to modelling mistakes are these mentioned [31].

$$elr(\mathtt{A[}\wedge\mathtt{]}) = \mathtt{A[}\vee\mathtt{]}$$
$$elr(\mathtt{A[}\vee\mathtt{]}) = \mathtt{A[}\wedge\mathtt{]}$$

Logical Operand. (eld) introduces mutations by replacing variable and expression logical operands with $true$ or $false$ constant values.

$$eld(\texttt{A[b]}, \texttt{true}) = \texttt{A[true]}$$
$$eld(\texttt{A[b]}, \texttt{false}) = \texttt{A[false]}$$

Arithmetic Operator. (ear) replaces a basic arithmetic operator with one of the three others. The four operators considered are sum, subtraction, multiplication and division.

$$ear(\texttt{A[op}_1\texttt{]}, \texttt{op}_2) = \texttt{op}_1, \quad \text{where } \texttt{op}_2 \in \{+, -, *, /\} \land \texttt{op}_1 \neq \texttt{op}_2 \bullet \texttt{P[op}_2\texttt{]}$$

Unary Insertion. (eur) inserts the minus modifier in front of an arithmetic expression. The variable e stands for an arithmetic expression in a process P.

$$eur(\texttt{A[e]}) = \texttt{A[}-\texttt{e]}$$

Add to Operand. (eak) increments an arithmetic operand by an integer constant k. The variable v stands for a numeric variable used in an expression in any schema or action.

$$eak(\texttt{A[v]}, \texttt{k}) = \texttt{A[(v + k)]}$$

Subtract from Operand. (esk) is similar, but subtracts the integer constant k.

$$esk(\texttt{A[v]}, \texttt{k}) = \texttt{A[(v} - \texttt{k)]}$$

Arithmetic Operand. (ead) arbitrarily replaces a numeric variable used as an arithmetic operand with another, keeping the types compatible. The variables v and u stand for numeric variables of the same type.

$$ead(\texttt{A[v]}, \texttt{u}) = \texttt{A[u]}$$

Relational Operator. (err) replaces any of the relational operators $<$, $\leq$, $>$, $\geq$, $=$, $\neq$ with any of the others from this same set.

$$err(\texttt{A[op}_1\texttt{]}, \texttt{op}_2) = \texttt{op}_1, \quad \text{where } \texttt{op}_2 \in \{<, \leq, >, \geq, =, \neq\} \land \texttt{op}_1 \neq \texttt{op}_2 \bullet \texttt{A[op}_2\texttt{]}$$

A mutation operator is *ideal* for a specification if it generates only mutants of interest; an ideal operator never leads to a refinement. Ideality is a strong requirement, not always achievable. None of the operators above is ideal. In fact, there can be no ideal operator for *Circus*: a single change to a process definition can never be guaranteed to lead to a non-refinement for every process. This can be seen by considering a process Q, an arbitrary mutation operator Op, and a process P defined as $P = Q \sqcap Op(Q)$. In applying Op to P, it is always possible to obtain $P' = Op(Q) \sqcap Op(Q)$. Properties of *Circus* guarantee that $P' = Op(Q)$, which does refine P.

4.3. Comparison with mutation testing based on LOTOS

Several specification languages make it possible to mix data type and behaviour descriptions. The approach followed above for the definition of a set of mutation operators can be easily transposed to them. We take as an example the full LOTOS

specification language, whose mutations have been studied in [47], and whose testing theory is developed in [25]. In a few words, LOTOS combines algebraic data type specifications and parameterised process definitions.

The LOTOS notation for process definition is syntactically close to those of *Circus* and CSP; however, there are semantic differences that arise, as far as testing is concerned, in the definition of refusals and acceptance sets. Given the definition of these sets, symbolic exhaustive test sets have been defined for $conf$ [24] and for the $ioco$ [11] conformance relation [25]. The main difference to *Circus* is the notion of state, which does not exist in LOTOS, and is simulated by process parameters like in CSP.

In [47], the authors propose a set of mutation operators. Most of them are not special to LOTOS and, as ours, have been taken or slightly adapted from [33] and [31]. They correspond to those indicated with source CSP or CSP* in Table 1, and seem to provide a kernel of mutation operators for specification languages of this category. The few specific operators defined in [47] and here are related to parameters in LOTOS, and to action and schema names in the case of *Circus*.

5. Tool support

To support *Circus* mutation testing and explore the practical aspects of our technique, we have implemented a prototype tool in Java to mutate specifications using the mutation operators of Section 4. Syntax-tree manipulation is provided by the CZT framework [48], using the *Circus* parser extension [49].

At the current stage, the prototype provides a simple command line interface that takes as inputs a mutation operator reference and a LaTeX *Circus* specification, according to the CZT style guide for *Circus*[1]. The output is presented as LaTeX *Circus* specifications, one for each mutation produced by the selected operator. In Table 2, we show the number of mutants produced by each operator when applied to $Chrono$ (in Figure 1). Operators not applicable to this example have not been considered.

Once the mutants are generated, we have to generate tests to kill them. We present below two approaches for the generation of mutant-killing test cases.

The first approach requires the use of a model checker. Similar approaches for generating test cases using model checkers are popular; see, for instance, [50] for a survey. In our case, the checker is used to establish whether a generated mutant is a refinement of the original specification and, if not, to yield some counterexamples that provide a basis for building killer tests as illustrated in Section 3.

For *Circus*, there is no mature refinement model checker available at this time. It is not the objective of our work to develop one (and there are ongoing efforts [51, 52] in this direction). However, for some *Circus* specifications, we can overcome this barrier via a translation of *Circus* models to CSP and the use of a well-known, mature refinement checker for CSP, FDR [53]. We detail this approach in Section 5.1.

The second approach for the generation of mutant-killing test cases is a (guided) generation of traces directly from the *Circus* specification. The traces are then con-

[1] Available at: `http://czt.sourceforge.net/latex/circus/circus-guide.pdf`, accessed: Dec/13/2015.

Abbreviation	Operator	Mutants	Killed by FDR
ped	Event Drop	1	1
per	Event Replacement	2	2
pei	Event Insert	2	2
pco	Choice Operator	2	2
ppoSeqPar	Sequential Composition	1	1
ppoSeqInt	Interleave	1	1
pmr	Message Replacement	2	2
pci	Communication Insert	1	1
pce	Communication Elimination	1	1
pprSchema	Schema Name	6	6
eniGuard	Guard Negation	2	2
eld	Operand Replacement	4	4
ear	Operator Replacement	1	1
eur	Unary Insertion	2	2
eak	Add k to Operand	6	6
esk	Sub k from Operand	6	6
ead	Operand Replacement	6	6
err	Operator Replacement	10	8

Table 2: Analysis of generated mutants for $Chrono$

verted into test cases. Various techniques can be used to guide the trace generation in order to obtain relevant traces. We describe this approach in Section 5.2.

5.1. Test generation via CSP and FDR

This approach is applicable only to *Circus* specifications with simple data models that can be directly encoded in CSP without further refinement. It has been, in any case, very useful to validate our ideas and approach. We discuss below the translation from *Circus* to CSP and the test-generation technique.

5.1.1. Translating Circus specifications into CSP

As shown in [54], a subset of *Circus* can be automatically translated to CSP; the main challenge is complex Z specifications. To carry out the experiments reported here and validate our approach to testing, however, we have carried out translation by hand.

To work with FDR, the specifications must be translated into the machine readable dialect of CSP [55] following the guidelines below:

- Channels and types are kept;
- The state components are encapsuled into a single data type with constructor $AState$, and passed as parameter to all processes and functions;
- A *Circus* action is translated into a CSP process;
- A reference to a schema s in an action A is unfolded to an intermediary process $A\_s(s(AState))$;

- Loops are translated as tail recursive calls;
- State changes defined by Z operations are performed by CSP functions from $AState$ to $AState$.

Example 6. *For instance, Figure 13 shows the translation of* $Chrono$*. The* $AInit$*,* $IncSec$ *and* $IncMin$ Circus *schemas are translated to the* `ainit`*,* `incsec` *and* `incmin` *functions. Arguments and results of these functions are yielded by the type constructor* `AState`*, encapsulating values for the two state components* sec *and* min*. The anonymous main action of* $Chrono$ *is translated into the* `CHRONO` *CSP process. The initialisation is captured by the parameter of the call to process* `RUN`*. The loop is captured by parameterised recursive calls at the end of each choice of* `RUN`*. The references to the schemas* $IncSec$ *and* $IncMin$ *inside the* Run *action are unfolded into the sub-process* `RUN_incsec` *and* `RUN_incsec_incmin`*.*

□

5.1.2. Using FDR for fault-based test-case generation

In Section 3 we have explained that counterexamples yielded by a refinement check of a mutant against the original specification provide bases to build tests to kill the mutant. Figure 14 shows the output of FDR for the check $CHRONO \sqsubseteq_F MCHRONO$ for failures refinement in the context of the definitions in Figure 13. It defines the following trace s, forbidden continuation a, and acceptance set X.

$$s = \langle tick, time \rangle \qquad a = out.1.1 \qquad X = \{out.0.1\}$$

For this experiment, the production of a counterexample gives rise to our account of the mutant as killed in Table 2. As stated in Theorem 1, finding such counterexamples is enough to assure that $MCHRONO$ is a mutant of interest: it gives us the necessary elements to build a test case for, at least, the failures fault-based set.

In the particular example of the $CHRONO \sqsubseteq_F MCHRONO$ analysis, we have both a failure and a forbidden continuation, allowing the construction of one test for each of the $FBTests_T^{CHRONO}(MCHRONO)$ and $FBTests_F^{CHRONO}(MCHRONO)$ sets. Such tests are constructed using the T_T and T_F functions.

$$T_T(s, a) = inc \rightarrow tick \rightarrow inc \rightarrow time \rightarrow pass \rightarrow out.1.1 \rightarrow fail \rightarrow \textbf{Stop}$$
$$T_F(s, X) = inc \rightarrow tick \rightarrow fail \rightarrow time \rightarrow fail \rightarrow out.0.1 \rightarrow pass \rightarrow \textbf{Stop}$$

In the traces of the execution of a parallel composition between the mutant $MCHRONO$ and any of these tests the last verdict event is $fail$, thus the tests kill the mutant.

All mutants generated for $Chrono$ (in Figure 1) have been translated to CSP and analysed with FDR. As shown in Table 2, only two of the mutants are not of interest. Both are generated by the relational operator replacement err, and are in Figure 15. The changes introduced by err in both $ChronoErrEquiv1$ and $ChronoErrEquiv2$ occur in the expressions guarding the external choice between $IncMin$ and $Skip$. In $ChronoErrEquiv1$, the original guard ($sec = 0$) is replaced with ($sec \leq 0$), and for $ChronoErrEquiv2$, the guard ($sec \neq 0$) is replaced with ($sec > 0$). Both changes cause no observable effect, since `Range` imposed lower boundary to sec is 0.

```
---- The chronometer in CSP_M
--Type and channels declaration
Range = {0..59}
channel tick, time
channel out:Range.Range

--State simulation with single data type
Minsec = {(min,sec) | min  <- Range, sec <- Range}
datatype Clock = AState.Minsec

--AInit schema translation: state initialization min=0 and sec=0
ainit() = AState.(0,0)
--IncSec schema translation: increments sec by one within Range
incsec(AState.(min,sec)) = AState.(min,(sec+1)%60)
--IncMin schema translation: increments min by one within Range
incmin(AState.(min,sec)) = AState.((min+1)%60,sec)

--Anonymous action to RUN process, with ainit state inialization
CHRONO = RUN(ainit())
--Run action translation, stateful loop achieved using recursion
RUN(AState.(min,sec)) =
   tick -> RUN_incsec(incsec(AState.(min,sec)))
   []
   time -> out.min.sec -> RUN(AState.(min,sec))
--Run sub-process translation, applying incsec function
RUN_incsec(AState.(min,sec)) =
   (sec == 0) & RUN_incsec_incmin(incmin(AState.(min,sec)))
   []
   (sec != 0) & RUN(AState.(min,sec))
RUN_incsec_incmin(AState.(min,sec)) = RUN(AState.(min,sec));

--First mutant generated with eniGuard operator
MCHRONO = MRUN(ainit())
MRUN(AState.(min,sec)) =
   tick -> MRUN_incsec(incsec(AState.(min,sec)))
   []
   time -> out.min.sec -> MRUN(AState.(min,sec))
MRUN_incsec(AState.(min,sec)) =
   --Negated guard mutation introduced below
   (not sec == 0) & MRUN_incsec_incmin(incmin(AState.(min,sec)))
   []
   (sec != 0) & MRUN(AState.(min,sec))
MRUN_incsec_incmin(AState.(min,sec)) = MRUN(AState.(min,sec));
```

Figure 13: *Chrono* in CSP

```
Result: Failed
Visited States: 7
Visited Transitions: 17
Visited Plys: 2
Estimated Total Storage: 67MB
Counterexample (Trace Counterexample)
   Specification Debug:
      Trace: <tick, time>
      Available Events: {out.0.1}
   Implementation Debug:
      MRUN(AState.(0, 0)) (Trace Behaviour):
         Trace: <tick, time>
         Error Event: out.1.1
```

Figure 14: FDR output for $CHRONO \sqsubseteq_F MCHRONO$

5.2. Test Generation from Circus specifications

The generation of tests from *Circus* specifications has been investigated and automated in the CirTA tool [56, 57], which uses an exhaustive approach to collect cstraces, along with their symbolic forbidden continuations and minimal acceptance sets, and enriches them with inc, $pass$ and $fail$ verdict events. Since in that work there is no selection based on the text of the specification, there is no need for specification traces.

As explained in Section 3, for generating mutant-killing test cases, a first selection step of relevant specification traces is needed. The mutant (or, more precisely, the part where it differs from the specification) serves as a guide for the collection of these traces. In this section, we elaborate on this idea, proposing a chain of tools for test generation following the approach sketched at the end of Section 3.

Figure 16 depicts our proposed tool chain. In the first step, the specification, the mutation point, and the mutant are provided as input to a specification-trace generator (SpTG), which derives specification traces using a guided symbolic execution of the mutant. The aim is to generate relevant specification traces, as defined in Section 3.4. For that, techniques of slicing can be used, similar to what is done for program analysis [58] and communicating automata specifications [59]. A generator based on the transition system that characterises specification traces is under development. The slicer and the check with respect to the original specification are the next steps.

We note that, instead of producing linear traces, the symbolic execution can easily be adapted for producing some symbolic execution tree. Such a tree can be the basis of tree-shaped tests, which avoid inconclusive verdicts by relaxing the constraint introduced by linear tests, that the SUT must follow one expected trace: the SUT can choose among the correct traces embedded in the tree. Such a factorization of linear tests, as discussed in [41], has the advantage of decreasing both the number of tests and the number of inconclusive verdicts, but the drawback that it leaves some control to the SUT with the risk that some relevant traces are not attempted.

$$
\begin{array}{l}
\textbf{process}\ \mathit{ChronoErrEquiv1} \mathrel{\widehat{=}} \\
\ldots \\
\quad \mathit{Run} \mathrel{\widehat{=}}\ (\mathit{tick} \rightarrow \mathit{IncSec};\ ((\mathit{sec} \leq 0)\ \&\ \mathit{IncMin}) \\
\qquad\qquad\qquad\qquad \Box((\mathit{sec} \neq 0)\ \&\ \mathit{Skip}))) \\
\qquad\qquad \Box(\mathit{time} \rightarrow \mathit{out}\,!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}) \\
\quad \bullet\ (\mathit{AInit};\ (\mu\ X \bullet (\mathit{Run};\ X)))
\end{array}
$$

$$
\begin{array}{l}
\textbf{process}\ \mathit{ChronoErrEquiv2} \mathrel{\widehat{=}} \\
\ldots \\
\quad \mathit{Run} \mathrel{\widehat{=}}\ (\mathit{tick} \rightarrow \mathit{IncSec};\ ((\mathit{sec} = 0)\ \&\ \mathit{IncMin}) \\
\qquad\qquad\qquad\qquad \Box((\mathit{sec} > 0)\ \&\ \mathit{Skip}))) \\
\qquad\qquad \Box(\mathit{time} \rightarrow \mathit{out}\,!(\mathit{min}, \mathit{sec}) \rightarrow \mathit{Skip}) \\
\quad \bullet\ (\mathit{AInit};\ (\mu\ X \bullet (\mathit{Run};\ X)))
\end{array}
$$

Figure 15: Two equivalent mutants of *Chrono* exposed by FDR

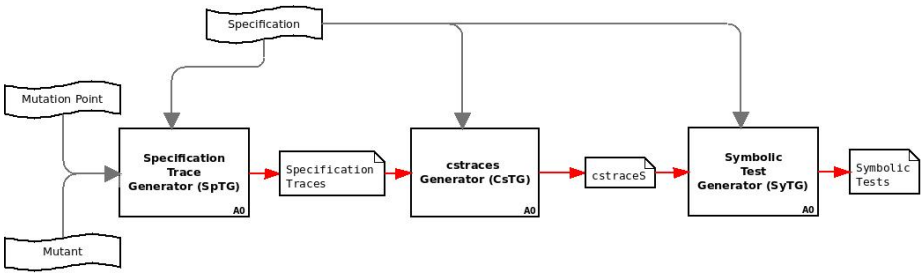

Figure 16: Test Generation from *Circus* specifications.

In a second step, the set of specification traces are given as input to a cstrace generator (CsTG). Each specification trace is converted into one cstrace. A constructive conversion procedure is formally specified in [28]. As explained in Section 3, it is based on an operational semantics that characterises the behavior of the path of a *Circus* specification identified by the labels of a specification trace. As can be expected, since the events, guards, and actions identified in specification traces are components of *Circus* actions, the operational semantics used in the conversion is very similar to a restricted subset of the *Circus* semantics. The automation of the conversion is, therefore, similar to the component of SpTG that calculates specification traces.

A prototype tool for proving (or disproving) refinements of *Circus* specifications, Isabelle/*Circus*, is presented in [60][2]. For each translated cstrace, CsTG checks whether it is a new cstrace or if it introduces any new deadlock, using Isabelle/*Circus* and reusing some components of the CirTA tool; if not, the cstrace is also discarded. CsTG

[2]Available at: `http://afp.sf.net/entries/Circus.shtml`, accessed: Dec/13/2015.

yields a set of cstraces with their symbolic forbidden continuations and acceptance sets. Each cstrace is converted (by SyTG) into a symbolic test by adding the verdicts at the appropriate points. These symbolic tests are adequately instantiated using an SMT solver as reported and illustrated in [57]. An example is given in Section 6.

The tool chain is expected to work using a lazy scheme, that is, a computation is only performed when its result is required. Thus, we avoid generating infinite sets of specification traces and cstraces, with, when necessary, some limitation on the length of the considered specification traces or cstraces.

The classical undecidable problem of equivalent mutants of programs [61] arises here under the form of checking whether a *Circus* model is a refinement of another one. For such a rich specification language, the use of a powerful proof assistant is unavoidable. The prototype Isabelle/*Circus* environment [60] is based upon the Isabelle/HOL proof environment where theories and rules of *Circus* semantics and refinement have been embedded. Once enriched with efficient dedicated proof tactics, it can be used both for a preliminary refinement check for test generation and for rejection of those cstraces that are cstraces of the original specification.

Our efforts are now focused in prototyping the necessary parts of the tool chain for automating the described strategy. The most complex and challenging tool is the SpTG, whose implementation starts to yield some preliminary results for a small subset of the *Circus* operational semantics. Considering the established theory background, it is a matter of time to achieve an operational level, allowing us to shift our concerns into issues of optimization and scalability.

6. Cash Machine: Another example

In this section, we illustrate how the approach we propose can be used for guiding test generation, using a more complex *Circus* specification as an example. We consider several mutant operators and the respective mutants, and discuss some interesting characteristics of the process of analysing them and generating the killer tests.

Figure 17 presents a slightly modified version of the cash-machine example used in [28]. First of all, we declare the set $CARD$ of valid cards. Next, we declare some channels. Requests for money are accepted by the cash machine through the channel inc, which takes a card and the amount to be withdrawn. The amount is a positive natural number. Cards are returned through a channel $outc$. The notes in and dispensed by the cash machine are those whose denominations are in the set $Note$. For simplicity, we consider just a few notes, and do not address the fact that the amount requested must be decomposable in terms of the notes available. If it is not, the machine fails to dispense the cash. In our model, cash is represented as a bag of notes: elements of the set $Cash$. If there is enough money in the machine and a way of providing the requested amount, the cash is output through a channel $cash$. The channel $refill$ is used to request the note bank of the cash machine to be refilled.

The cash machine accepts requests for cash and decides whether the cash should be dispensed. The only state component of $CashMachine$ is a function $nBank$ that records, for each denomination, the amount of notes available. The state is defined by a schema, $CMState$, which declares $nBank$ as a total function.

$[CARD]$
channel $inc : CARD \times \mathbb{N}_1;\ \ outc : CARD;\ cash : Cash;\ refill$
$Note == \{10, 20, 50\}$
$Cash ==$ bag $Note$

process $CashMachine \mathrel{\widehat=}$ **begin**

state $CMState == [\ nBank : Note \pfun 0 \,..\, cap\ ]$

DispenseNotes

$\Delta CMState$
$a? : \mathbb{N}_1$
$notes! : Cash$

$\Sigma\, notes! = a?$
$\forall\, n : Note \bullet (notes!\ \sharp\ n) \leq nBank\ n \land nBank'\ n = (nBank\ n) - (notes!\ \sharp\ n)$

DispenseError

$\Xi CMState$
$a? : \mathbb{N}_1;\ notes! : Cash$

$\neg\ \exists\, ns : Cash \bullet \Sigma\, ns = a? \land \forall\, n : Note \bullet (ns\ \sharp\ n) \leq nBank\ n$
$notes! = [\![\ ]\!]$

$Dispense == DispenseNotes \lor DispenseError$

$$\bullet \left(\begin{array}{l} \mu\, X \bullet \left(\begin{array}{l} inc?c?a \rightarrow \\ \left(\begin{array}{l} \quad outc!c \rightarrow X \\ \sqcap \\ \left(\begin{array}{l} \textbf{var}\ notes : Cash \bullet \\ \quad \left(\begin{array}{l} Dispense; \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} (notes \neq [\![\ ]\!]) \ \&\ cash!notes \rightarrow \textbf{Skip} \\ \sqcap \\ (notes = [\![\ ]\!]) \ \&\ \textbf{Skip} \end{array} \right) \end{array} \right) \end{array} \right) \end{array} \right) ; outc!c \rightarrow X \end{array} \right) \\ \sqcap \\ refill \rightarrow nBank := \{\, 10 \mapsto cap, 20 \mapsto cap, 50 \mapsto cap \,\}\ ;\ \ X \end{array} \right) \end{array} \right)$$

end

Figure 17: Cash machine specification

Abbreviation	Operator	Mutants
ped	Event Drop	1
pei	Event Insert	1
pco	Choice Operator	2
ppoSeqPar	Sequential Composition	3
ppoSeqInt	Interleave	3
pci	Communication Insert	4
pce	Communication Elimination	4
pprSchema	Schema Name	2
eni	Negation Insert	6
eniGuard	Guard Negation	2
elr	Operator Replacement	3
eld	Operand Replacement	16
ear	Operator Replacement	3
eur	Unary Insertion	1
eak	Add k to Operand	8
esk	Sub k from Operand	8
ead	Operand Replacement	6
err	Operator Replacement	20

Table 3: Count of mutants generated for $CashMachine$

The $DispenseNotes$ data operation takes an amount $a?$ as input and produces a bag of notes $notes!$ as output; it also updates $nBank$. It is defined using a Z schema that specifies a relation on $CMState$. The value of $notes!$ is nondeterministically chosen: it is any bag $notes!$ whose sum $\Sigma\, notes!$ of its elements is equal to $a?$, and such that, for each note denomination n, the number $notes!\ \sharp\ n$ of occurrences of n is less than or equal to the number $nBank\ n$ of notes of denomination n in the bank.

If there is no such bag, we have an error: the output is the empty bag $|[\,]|$, and the state is not changed. This is defined by the schema $DispenseError$. The total operation to $Dispense$ cash is the schema disjunction of $DispenseNotes$ and $DispenseError$.

The main action of $CashMachine$ defines that it accepts a request $inc?c?a$; this is an input of any card c and any amount a. It then decides whether to output the card using $outc$ and no money, or dispense the requested amount using $cash$. The decision is nondeterministic; it is defined by factors outside of this model: status of the card, balance on the corresponding account, and so on. The cash machine also accepts requests to *refill* the note bank. A recursion offers these choices over and over again.

We have used our mutant-generation prototype for the $CashMachine$ example. The resulting numbers are shown in Table 3. We consider some mutations below.

6.1. Communication elimination (pce) of the first occurrence of $outc!c$

We consider first the mutant obtained by applying the pce operator to remove the first occurrence of $outc!c$. We call $MutatedCashMachine'$ the resulting process. Among its specification traces that reach the mutation point, there are $\langle inc?c?a, refill\rangle$

and $\langle inc?c?a, inc?c?a\rangle$. The corresponding cstraces are shown below.

$(\langle \texttt{inc}?\alpha_{\emptyset}?\alpha_1, \texttt{refill}\rangle, \alpha_{\emptyset} \in \texttt{CARD} \land \alpha_1 \in \mathbb{N}_1)$
$(\langle \texttt{inc}?\alpha_{\emptyset}?\alpha_1, \texttt{inc}?\alpha_2?\alpha_3\rangle, \alpha_{\emptyset} \in \texttt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \alpha_2 \in \texttt{CARD} \land \alpha_3 \in \mathbb{N}_1)$

They are not cstraces of $CashMachine$, and so these traces of $MutatedCashMachine'$ are relevant for testing against traces refinement. The test generation can be done via the selection of one of the specification traces above, its translation into its cstrace, and the construction of the corresponding symbolic test. For the second one we have:

$inc \rightarrow inc?\alpha_0?\alpha_1 : (\alpha_0 \in CARD \land \alpha_1 \in \mathbb{N}_1) \rightarrow pass \rightarrow$
$inc?\alpha_2?\alpha_3 : (\alpha_2 \in CARD \land \alpha_3 \in \mathbb{N}_1) \rightarrow fail \rightarrow \textbf{Stop}$

By resolution of the constraints on $\alpha_0, \alpha_1, \alpha_2, \alpha_3$, choosing cards α_0 and α_2, and amounts α_1 and α_3, we can obtain a concrete test that kills $MutatedCashMachine'$.

6.2. Communication elimination (pce) of the second occurrence of $outc!c$

When removing the second occurrence of the $outc!c$ event, the following specification traces appear, that reach the mutation point:

$\langle \texttt{inc?c?a}, \texttt{var notes}, \texttt{Dispense}, \texttt{notes} \neq [\![\,]\!], \texttt{cash!notes}, \textbf{Skip}, \texttt{inc?c?a}\rangle$
$\langle \texttt{inc?c?a}, \texttt{var notes}, \texttt{Dispense}, \texttt{notes} \neq [\![\,]\!], \texttt{cash!notes}, \textbf{Skip}, \texttt{refill}\rangle$
$\langle \texttt{inc?c?a}, \texttt{var notes}, \texttt{Dispense}, \texttt{notes} = [\![\,]\!], \textbf{Skip}, \texttt{inc?c?a}\rangle$
$\langle \texttt{inc?c?a}, \texttt{var notes}, \texttt{Dispense}, \texttt{notes} = [\![\,]\!], \textbf{Skip}, \texttt{refill}\rangle$

The corresponding cstraces are shown below.

($\langle \texttt{inc}?\alpha_{\emptyset}?\alpha_1, \texttt{cash}!\alpha_2, \texttt{inc}?\alpha_3?\alpha_4\rangle$,
$\alpha_0 \in CARD \land \alpha_1 \in \mathbb{N}_1 \land \alpha_2 \in Cash \land \Sigma\alpha_2 = \alpha_1 \land \alpha_3 \in CARD \land \alpha_4 \in \mathbb{N}_1$)
($\langle \texttt{inc}?\alpha_{\emptyset}?\alpha_1, \texttt{cash}!\alpha_2, \texttt{refill}\rangle$,
$\alpha_0 \in CARD \land \alpha_1 \in \mathbb{N}_1 \land \alpha_2 \in Cash \land \Sigma\alpha_2 = \alpha_1$)
($\langle \texttt{inc}?\alpha_{\emptyset}?\alpha_1, \texttt{inc}?\alpha_2?\alpha_3\rangle, \alpha_{\emptyset} \in \texttt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \alpha_2 \in \texttt{CARD} \land \alpha_3 \in \mathbb{N}_1$)
($\langle \texttt{inc}?\alpha_{\emptyset}?\alpha_1, \texttt{refill}\rangle, \alpha_{\emptyset} \in \texttt{CARD} \land \alpha_1 \in \mathbb{N}_1$)

They are not cstraces of the original specification and can be used as bases for killer tests. The last two cstraces are the same as those obtained when dealing with the removal of the first occurrence of $outc!c$ in the previous section.

6.3. Substitution (pprSchema) of the reference to Dispense by DispenseError

As another example, we consider the mutant obtained by the application of the $pprSchema$ operator to the second occurrence of the $Dispense$ schema name (that is, the use of $Dispense$ in the variable block that declares $notes$ in the main action of the process $CashMachine$ in Figure 17). We can use $pprSchema$ to replace $Dispense$ with, for instance, $DispenseError$, whose precondition may not hold.

The specification trace below reaches the mutation point.

$\langle inc?c?a, \texttt{var}\ notes, \texttt{DispenseError}\rangle$

It is a specification trace of the mutated specification because, in some cases, the pre-

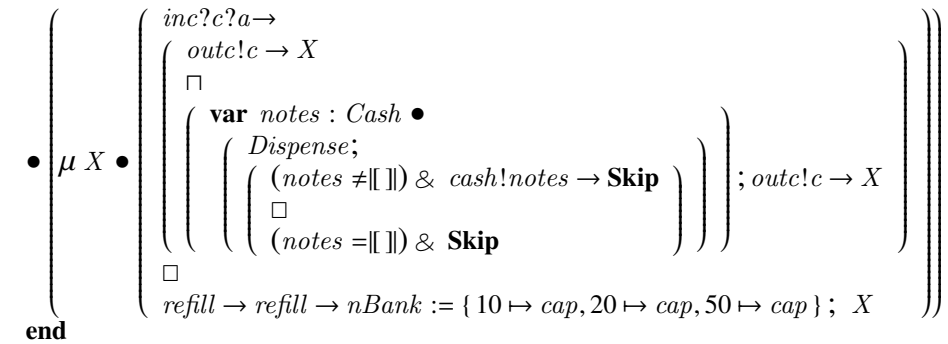

Figure 18: Mutated main action of $CashMachine$ with event $refill$ inserted/duplicated

condition of $DispenseError$ is satisfied: it depends on the initialisation of $nBank$.

The corresponding cstrace is:

$$(\langle \mathtt{inc}?\alpha_0?\alpha_1 \rangle, \alpha_0 \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1)$$

It is a cstrace of the original specification, but after $inc?c?a$, the internal choice leads either to $(outc!c)$ or to $DispenseError$. If the precondition of $DispenseError$ does not hold, it behaviour is divergent. Therefore, this mutant is discarded.

6.4. Insertion of event (pei) refill

The result of a mutation using pei to duplicate the event $refill$ in the main action of $CashMachine$ is given in Figure 18. It has new specification traces, like, for instance:

$$\langle \mathtt{refill}, \mathtt{refill}, \mathtt{nBank} := \{\, \mathtt{10} \mapsto \mathtt{cap}, \mathtt{20} \mapsto \mathtt{cap}, \mathtt{50} \mapsto \mathtt{cap} \,\} \rangle$$

The corresponding cstrace is $(\langle \mathtt{refill}, \mathtt{refill} \rangle, \mathbf{True})$, which is a cstrace of the original $CashMachine$ process. We observe the same situation for all the new specification traces: they are new because they embed the subtrace

$$\langle \ldots refill, refill, nBank := \{\, 10 \mapsto cap, 20 \mapsto cap, 50 \mapsto cap \,\}, \ldots \rangle$$

or finish with a double occurrent of $refill$. In the corresponding cstraces, these give rise to subtraces $(\langle \ldots \mathtt{refill}, \mathtt{refill}, \ldots \rangle, \ldots)$, which are admitted by $CashMachine$. This means that the original and the mutated processes have the same traces, and we cannot distinguish them with a test for traces inclusion.

This mutant, however, has new failures. For example,

$$\langle \mathtt{refill}, \mathtt{nBank} := \{\, \mathtt{10} \mapsto \mathtt{cap}, \mathtt{20} \mapsto \mathtt{cap}, \mathtt{50} \mapsto \mathtt{cap} \,\}, \mathtt{inc?c?a} \rangle,$$

is a specification trace of $CashMachine$, but not of the mutated process. After the trace $\langle refill \rangle$, the only accepted event of the mutated process is $refill$ and any instantiation

of $inc?c?a$ is refused. So, $\langle refill \rangle$ paired with a singleton set containing an event $inc.c.a$ is a failure of the mutated process, but not of $CashMachine$. On the other hand, the singleton set is a minimal acceptance of $CashMachine$ after $\langle refill \rangle$, and any instantiation of the symbolic test below is a killer test.

$$inc \rightarrow refill \rightarrow fail \rightarrow inc?\alpha_0?\alpha_1 : (\alpha_0 \in CARD \wedge \alpha_1 \in \mathbb{N}_1) \rightarrow pass \rightarrow \textbf{Stop}$$

Instantiation requires the choice of values for α_0 and α_1 that satisfy the given constraint.

6.5. Mutation of choice operator (pco)

A mutation of the external choice in the main action of $CashMachine$ into an internal choice, using pco, gives rise to a situation similar to that of the previous section. As noted in Section 4, such a mutation does not introduce new specification traces. It can, however, introduce failures. For instance, $refill$ may be refused after the empty trace by the mutated process, but not by $CashMachine$, which has a minimal $acceptance$ $\{refill\}$ after the empty trace. A killer test is $fail \rightarrow refill \rightarrow pass \rightarrow$ **Stop**.

6.6. Communication insert (pci)

The communication insert operator pci is similar to the event insert pei operator considered in the previous section. It is, however, applied to a communication instead of a synchronization. We consider here the duplication of $cash!notes$. A specification trace that reaches the mutation point is as follows.

$$\langle \texttt{inc?c?a}, \texttt{var notes}, \texttt{Dispense}, \texttt{notes} \neq [\![\,]\!], \texttt{cash!notes}, \texttt{cash!notes} \rangle$$

The corresponding cstrace is

$$\begin{array}{l} (\ \langle inc?\alpha_0?\alpha_1, cash!\alpha_2, cash!\alpha_3 \rangle, \\ \quad \alpha_0 \in CARD \wedge \alpha_1 \in \mathbb{N}_1 \wedge \alpha_2 \in Cash \wedge \Sigma\alpha_2 = \alpha_1 \wedge \alpha_3 \in Cash \wedge \alpha_3 = \alpha_2\ ) \end{array}$$

It is not a cstrace of $CashMachine$ and leads to the symbolic test below.

$$\begin{array}{l} inc \rightarrow inc?\alpha_0?\alpha_1 : (\alpha_0 \in CARD \wedge \alpha_1 \in \mathbb{N}_1) \rightarrow inc \rightarrow \\ cash!\alpha_2 : (\alpha_2 \in Cash \wedge \Sigma\alpha_2 = \alpha_1) \rightarrow pass \rightarrow \\ cash!\alpha_3 : (\alpha_3 \in Cash \wedge\ \ \alpha_3 = \alpha_2) \rightarrow fail \rightarrow \textbf{Stop} \end{array}$$

Once the above test is instantiated, it yields a killer test.

6.7. Mutation of sequential compositions (ppoSeqPar)

The new parallel compositions introduced by $ppoSeqPar$ synchronise on all channels in a given set CS and have no write access to any variables. In the main action of $CashMachine$, there are three sequential compositions. We show in Figure 19 the result of mutating the one insider the variable block that declares $notes$.

In this case, the mutated process may deadlock after $Dispense$. Both parallel actions act on the arbitrary initial value of $notes$. If that value is not the empty bag, since the synchronisation on $cash$ required by the external choice cannot happen, because $Dispense$ is not ready to communicate on $cash$, there is a deadlock. If, however, that

$$
\left(\begin{array}{l}
\mu\, X \bullet \left(\begin{array}{l}
inc?c?a \rightarrow \\
\left(\begin{array}{l}
\left(\begin{array}{l}
outc!c \rightarrow X \\
\sqcap \\
\left(\begin{array}{l}
\textbf{var}\ notes : Cash \bullet \\
\left(\begin{array}{l}
Dispense \\
\quad [\![\{\!| inc, outc, cash, refill |\!\}]\!] \\
\left(\begin{array}{l}
(notes \neq [\![\,]\!]) \ \&\ cash!notes \rightarrow \textbf{Skip} \\
\sqcap \\
(notes = [\![\,]\!]) \ \&\ \textbf{Skip}
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}\right) ; outc!c \rightarrow X \\
\sqcap \\
refill \rightarrow nBank := \{\, 10 \mapsto cap, 20 \mapsto cap, 50 \mapsto cap \,\} \ ;\ X
\end{array}\right)
\end{array}\right)
\end{array}\right)
$$

Figure 19: Mutation of the first occurrence of ; by $ppoSeqPar$

arbitrary initial value happens to be the empty bag, the mutated process does not deadlock. It dispenses the card via $outc$ without changing the state, since the changes made by the parallel actions cannot be recorded in any variables. So, we have a fail safe.

The cstrace $(\langle \mathtt{inc}?\alpha_{\emptyset}?\alpha_1 \rangle, \alpha_{\emptyset} \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1)$ is associated with a set of symbolic acceptances including $(\mathtt{outc}!\alpha_3, \alpha_{\emptyset} \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \alpha_3 = \alpha_{\emptyset})$ and $(\mathtt{cash}!\alpha_3, \alpha_{\emptyset} \in \mathtt{CARD} \land \alpha_1 \in \mathbb{N}_1 \land \alpha_3 \in \mathtt{Cash} \land \Sigma\alpha_3 = \alpha_1)$. This is not an acceptance of the mutated process, and can be used to define a killer test.

$$
\begin{array}{l}
inc \rightarrow inc?\alpha_0?\alpha_1 : (\alpha_0 \in CARD \land \alpha_1 \in \mathbb{N}_1) \rightarrow fail \rightarrow \\
\left(\begin{array}{l}
outc!\alpha_3 : (\alpha_0 \in CARD \land \alpha_1 \in \mathbb{N}_1 \land \alpha_3 = \alpha_0) \rightarrow pass \rightarrow \textbf{Stop} \\
\sqcap \\
cash!\alpha_3 : (\alpha_0 \in CARD \land \alpha_1 \in \mathbb{N}_1 \land \alpha_3 \in Cash \land \Sigma\alpha_3 = \alpha_1) \rightarrow pass \rightarrow \textbf{Stop}
\end{array}\right)
\end{array}
$$

Instantiation keeps the restrictions on the inputs and defines specific values for outputs. Above, the value of α_3 has different constraints in the outputs via $outc$ and $cash$.

Based on the examples in this section, we note that the construction of test cases from mutants is a very challenging task, which demands the support of specialised tools. The tool support we discuss in Section 5 is paramount for making our approach practical and we are placing efforts in developing it with the aid of frameworks for symbolic manipulation and constraint solvers. In Appendix B, we include further examples of mutants for a *Circus* specification (of an Emergency Recovery System).

7. Related Work

Mutation has already been used for guiding test-case generation from (formal) models in many different pieces of work [62, 63, 64, 31, 36, 65, 66, 67, 68]. As already mentioned, it is also used to assess the quality of a test suite. The possibility of selecting on which errors to concentrate is a good mechanism to tackle the explosion of test cases (see [50] and [66] for recent surveys). A more general survey on the main developments of mutation testing is found in [69].

Budd and Gopal [70] have pioneered the investigation on testing by mutating specifications. They propose the use of a mutant of a specification to generate tests for programs. Specifications are based on predicate calculus. The mutant is a variation of a predicate defining the expected behavior of the program. A test case is an input for which the original specification and the mutant produce different truth values.

Ammann et al. [71] use the SMV model checker to generate test cases from mutants of a specification. The mutant operators are defined at the syntactical level, and the test cases are traces of the specification but not of the mutant. The application of a similar approach to an industrial case study is reported in [72]. Simulink models are mutated and the test cases are generated with CBMC (bounded model checker for C). Generation of tests from mutated Simulink models is also investigated in [73].

The work of Papadakis et al. [74, 75, 76] automates white-box test-case generation for programs, relying on symbolic and concolic execution, mutant schemata and the weak mutation-testing criterion. The goals are to lower the cost of test-case generation and increase the quality of the obtained test suite.

Mutation testing has also been used to generate test cases for security-critical systems in [63]. The mutants are used to model vulnerabilities. A constraint solver is employed to find a trace of the mutant that does not satisfy the security properties of the system. If such a trace exists, the mutant introduces a vulnerability and the test case shows how to exploit it. This approach is similar to ours, but due to the nature of the model we use, refinement verification is employed instead of a constraint solver. Similarly, the work by John Clark et al. [31], whose mutant operators we have considered in Section 4, uses mutation testing for checking system security. Moreover, Clark et al. use the mutants to validate the specification, while we use them to generate tests.

Krenn and Aichernig [64] mutate program contracts and use the SAT solvers Boogie and Z3 to generate test cases for Spec# models. Their proposed technique automatically generate tests that can distinguish whether an implementation refines a faulty specification. We use mutants to select tests from our fault-based exhaustive test sets.

Aichernig and colleagues have advanced the application of mutation testing for models in various formalisms [36, 65, 62, 66]. A test case is seen as an abstraction (according to a traces-refinement relation) of the specification, that is, an implementation (or specification) should refine the test case if it passes the test. Thus, test-case synthesis is a reverse refinement problem. Mutants guide the generation of test cases; a fault detecting test case is an abstraction of the specification, but not of the mutant. The theory is developed in the context of Hoare and He's UTP [19]. Mutants are generated both for the specification and the implementation. Our approach is on the same vein of Aichernig and colleagues' work, extending the main concepts to a new formalism.

8. Conclusions and future work

We have formally defined mutation testing for *Circus* by characterising (1) the exhaustive sets of tests that can kill a given mutant, considering both traces refinement and $conf$, that is, process refinement in *Circus* as a whole; (2) a technique to select tests from these sets based on specification traces of the mutants; and (3) an extensive collection of operators to generate mutants considering faults related to both reactive

and data manipulation behaviour. Like the *Circus* testing theory, this work is of general relevance for state-rich algebras for refinement.

To make our ideas concrete, we have led some preliminary experimentation on models that are both simple enough (regarding its data structures) and finite to enable model checking via FDR. We have also developed a first tool in Java for mutant generation, which is the front-end both for using FDR and for a chain of tools we have defined for mutants analysis and killer-tests generation. The mutant analysis can make use of the Isabelle*Circus* refinement checker, and the test generation can use some components of the exhaustive test generator CirTA.

We, however, are not in a position to provide experimental results that address scalability, since the tool chain we have available so far is not suitable for conducting such studies. The main bottleneck we foresee is the cost of the symbolic manipulation of the model, which we expect to tackle using slicing techniques. Another issue is related to decidability problems regarding the identification of useless mutants, namely, those that refine the original model. We have not addressed this yet, but will in the future steps of our research using model checking and automated theorem proving.

Given a mutation operator such as one of those given in Section 4, it is attractive to avoid constructing the mutants to generate the tests to kill them as described above. For that, we need a way of calculating the traces and failures of the mutants without constructing them. This calculation could take advantage of the knowledge of traces and failures of the original process, but the effect of the mutation operators on the semantics of a process is not direct. More precisely, the semantic models of interest are not a congruence for the mutant operators. This gives rise to an interesting challenge.

Generally, mutations are syntactic changes. Since our specification traces are close to the syntax of the specifications, it may well be the case that they do provide an adequate way to construct traces of the mutants in terms of those of the original specification. An analysis of mutation operators for specification traces and their relation to our operators is an interesting avenue for future work.

We also plan to investigate "semantic mutations". The idea is to consider the sets of traces and failures of the specification and to study mutations of these sets. This will avoid the construction of the syntactic mutants. It is important to note that the mutations must preserve the properties of the sets, for instance, the set of traces is prefix-closed, and the set of failures is subset-closed with respect to refusal sets.

Acknowledgments

We are grateful to the Digiteo research cluster for their financial support of our collaboration (Convention N2014-1411D), to the Brazilian Funding Agency CNPq (Grant 400834/2014-6), and to the Royal Society.

References

[1] T. S. Chow, Testing software design modeled by finite-state machines, IEEE Trans. Softw. Eng. 4 (3) (1978) 178–187.

[2] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, A. Ghedamsi, Test selection based on finite state models, IEEE Trans. Softw. Eng. 17 (6) (1991) 591–603.

[3] A. Petrenko, N. Yevtushenko, Testing from partial deterministic FSM specifications, IEEE Trans. on Computers 54 (9).

[4] A. Petrenko, G. v. Bochmann, M. Yao, On fault coverage of tests for finite state specifications, Computer Networks and ISDN Systems 29 (1) (1996) 81–106.

[5] R. Dorofeeva, K. El-Fakih, N. Yevtushenko, An improved conformance testing method, in: Formal Techniques for Networked and Distributed Systems, Vol. 3731 of LNCS, Springer, 2005, pp. 204–218.

[6] R. M. Hierons, H. Ural, Optimizing the length of checking sequences, IEEE Trans. on Computers 55 (5) (2006) 618–629.

[7] G.-V. Jourdan, H. Ural, H. Yenigün, D. Zhu, Using a SAT solver to generate checking sequences, in: 24th Int. Symp. on Computer and Information Sciences, ISCIS, 2009, pp. 549–554.

[8] R. M. Hierons, G.-V. Jourdan, H. Ural, H. Yenigün, Checking sequence construction using adaptive and preset distinguishing sequences, in: 7th IEEE Int. Conf. on Sof. Eng. and Formal Methods, SEFM 2009, 2009, pp. 157–166.

[9] Q. Guo, R. M. Hierons, M. Harman, K. Derderian, Heuristics for fault diagnosis when testing from finite state machines, Softw. Test., Verif. Reliab. 17 (1) (2007) 41–57.

[10] M. Gromov, K. El-Fakih, N. Shabaldina, N. Yevtushenko, Distinguing nondeterministic timed finite state machines, in: Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings, 2009, pp. 137–151.

[11] J. Tretmans, Test generation with inputs, outputs, and quiescence., in: TACAS'96, Vol. 1055 of LNCS, Springer, 1996, pp. 127–146.

[12] M. Van Der Bijl, A. Rensink, J. Tretmans, Compositional testing with ioco, Formal Approaches to Software Testing (2004) 1102–1102.

[13] M. Weiglhofer, F. Wotawa, Asynchronous input-output conformance testing, in: COMPSAC'09, IEEE, 2009, pp. 154–159.

[14] M. Weiglhofer, B. K. Aichernig, Unifying input output conformance, in: Unifying Theories of Programming, Second International Symposium, UTP 2008, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers, Springer, 2008, pp. 181–201.

[15] M. Krichen, A formal framework for black–box conformance testing of distributed real–time systems, Int. Jal of Critical Computer-Based Systems 3 (1) (2012) 26–43.

[16] J. C. P. Woodcock, J. Davies, Using Z—Specification, Refinement, and Proof, Prentice-Hall, 1996.

[17] A. W. Roscoe, The Theory and Practice of Concurrency, Prentice-Hall Series in Computer Science, Prentice-Hall, 1998.

[18] C. C. Morgan, Programming from Specifications, 2nd Edition, Prentice-Hall, 1994.

[19] C. A. R. Hoare, H. Jifeng, Unifying Theories of Programming, Prentice-Hall, 1998.

[20] A. L. C. Cavalcanti, P. Clayton, C. O'Halloran, From Control Law Diagrams to Ada via *Circus*, Formal Aspects of Computing 23 (4) (2011) 465–512.

[21] L. Freitas, J. P. McDermott, Formal methods for security in the xenon hypervisor, Int. Jal on Software Tools for Technology Transfer, 13 (5) (2011) 463 – 489.

[22] A. L. C. Cavalcanti, M.-C. Gaudel, Testing for Refinement in *Circus*, Acta Informatica 48 (2) (2011) 97–147.

[23] M.-C. Gaudel, Testing can be formal, too, in: Int. Joint Conf., Theory And Practice of Software Development, Vol. 915 of LNCS, Springer, 1995, pp. 82–96.

[24] M.-C. Gaudel, P. J. James, Testing algebraic data types and processes : a unifying theory, Formal Aspects of Computing 10 (5-6) (1998) 436–451.

[25] G. Lestiennes, M.-C. Gaudel, Testing processes from formal specifications with inputs, outputs, and datatypes, in: IEEE Int. Symp.on Software Reliability Engineering, ISSRE, 2002, pp. 3–14.

[26] J. C. P. Woodcock, A. L. C. Cavalcanti, M.-C. Gaudel, L. J. S. Freitas, Operational Semantics for *Circus*, Formal Aspects of ComputingTo appear.

[27] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, Unifying Theories in ProofPowerZ, Formal Aspects of Computing, online firstDOI 10.1007/s00165-007-0044-5.

[28] A. L. C. Cavalcanti, M.-C. Gaudel, Data Flow coverage for *Circus*-based testing, in: FASE 2014, Vol. 8441 of LNCS, 2014, pp. 415–429.

[29] M. Papadakis, N. Malevris, Automatic mutation test case generation via dynamic symbolic execution, in: Software reliability engineering (ISSRE), IEEE, 2010, pp. 121–130.

[30] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Software Eng. 37 (5) (2011) 649–678.

[31] T. Srivatanakul, J. A. Clark, S. Stepney, F. Polack, Challenging formal specifications by mutation: a CSP security example, in: 10th Asia-Pacific Software Engineering Conference, IEEE Press, 2003, pp. 340–350.

[32] D. R. Kuhn, Fault Classes and Error Detection Capability of Specification-Based Testing, ACM Transactions on Software Engineering and Methodology 8 (4) (1999) 411–424.

[33] P. E. Black, V. Okun, Y. Yesha, Mutation operators for specifications, in: Automated Software Engineering. ASE 2000, IEEE, 2000, pp. 81–88.

[34] C. Fischer, Combination and Implementation of Processes and Data: from CSP-OZ to Java, Ph.D. thesis, Fachbereich Informatik Universität Oldenburg (2000).

[35] S. Schneider, H. Treharne, CSP Theorems for communicating B machines, Formal Aspects of Computing 17 (4) (2005) 390–422.

[36] B. Aichernig, H. Jifeng, Mutation testing in UTP, Formal Aspects of Computing 21 (2008) 3364.

[37] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, A UTP Semantics for *Circus*, Formal Aspects of Computing 21 (1-2) (2009) 3–32.

[38] J. C. King, Symbolic execution and program testing, Commun. ACM 19 (7) (1976) 385–394.

[39] E. Brinksma, A theory for the derivation of tests, in: Protocol Specification, testing and Verification VIII, North-Holland, 1988, pp. 63–74.

[40] A. L. C. Cavalcanti, M.-C. Gaudel, A note on traces refinement and the $conf$ relation in the Unifying Theories of Programming, in: Unifying Theories of Programming 2008, Vol. 5713 of LNCS, Springer, 2010, pp. 42–61.

[41] A. L. C. Cavalcanti, M.-C. Gaudel, Testing for Refinement in CSP, in: 9th ICFEM, Vol. 4789 of LNCS, Springer, 2007, pp. 151–170.

[42] S. Fujiwara, G. v. Bochmann, Testing non-deterministic state machines with fault coverage, in: IFIP TC6/WG6.1 4th Int. Wshop on Protocol Test Systems IV, North-Holland, 1991, pp. 267–280.

[43] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, FDR3 A Modern Refinement Checker for CSP, in: Tools and Algorithms for the Construction and Analysis of Systems, 2014, pp. 187–201.

[44] J. Huo, A. Petrenko, Transition covering tests for systems with queues, Softw. Test., Verif. Reliab. 19 (1) (2009) 55–83.

[45] A. L. C. Cavalcanti, M.-C. Gaudel, Specification Coverage for Testing in *Circus*, in: Unifying Theories of Programming, Vol. 6445 of LNCS, Springer, 2010, pp. 1–45.

[46] M. V. M. Oliveira, Formal Derivation of State-Rich Reactive Programs Using *Circus*, Ph.D. thesis, University of York (2006).

[47] B. K. Aichernig, C. C. Delgado, From faults via test purposes to test cases: On the fault-based testing of concurrent systems, in: FASE 2006, Vol. 3922 of LNCS, Springer, 2006, pp. 324–338.

[48] P. Malik, M. Utting, CZT: A framework for Z tools, in: ZB 2005: Formal Specification and Development in Z and B, Vol. 3455 of LNCS, Springer, 2005, pp. 65–84.

[49] T. Miller, L. Freitas, P. Malik, M. Utting, CZT support for Z extensions, in: Integrated Formal Methods, Vol. 3771 of LNCS, Springer, 2005, pp. 227–245.

[50] G. Fraser, F. Wotawa, P. E. Ammann, Testing with model checkers: a survey, Software Testing, Verification and Reliability 19 (3) (2009) 215–261.

[51] L. J. S. Freitas, Model Checking *Circus*, Ph.D. thesis, University of York, Department of Computer Science (2006).

[52] A. Mota, A. Farias, A. Didier, J. Woodcock, Rapid prototyping of a semantically well founded Circus model checker, in: Software Engineering and Formal Methods, Vol. 8702 of LNCS, Springer, 2014, pp. 235–249.

[53] Formal Systems (Europe) Ltd, FDR: User Manual and Tutorial, version 2.28 (1999).

[54] M. V. M. Oliveira, A. C. Sampaio, M. S. Conserva Filho, Model-checking Circus state-rich specifications, in: Integrated Formal Methods, Vol. 8739 of LNCS, Springer, 2014, pp. 39–54.

[55] B. Scattergood, P. Armstrong, CSPM: A Reference Manual (Jan. 2011).

[56] A. Feliachi, Semantics-based testing for circus, Ph.D. thesis, LRI, Universite de Paris-Sud (2012).

[57] A. Feliachi, M.-C. Gaudel, B. Wolff, Symbolic test-generation in HOL-TestGen/Cirta: A case study, Int. J. Software Informatics 9 (2) (2015) 177–203.

[58] R. M. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Softw. Test., Verif. Reliab. 9 (4) (1999) 233–262.

[59] S. Labbé, J.-P. Gallois, Slicing communicating automata specifications: polynomial algorithms for model reduction, Formal Aspects of Computing 20 (6) (2008) 563–595.

[60] A. Feliachi, M. Gaudel, B. Wolff, Isabelle/Circus: A process specification and verification environment, in: Verified Software: Theories, Tools, Experiments -VSTTE 2012, Vol. 7152 of LNCS, Springer, 2012, pp. 243–260.

[61] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation, IEEE Trans. Softw. Eng. 40 (1) (2014) 23–42.

[62] B. K. Aichernig, Mutation testing in the refinement calculus, Formal Aspects of Computing 15 (2003) 280–295.

[63] G. Wimmel, J. Jurjens, Specification-based test generation for security-critical systems using mutations, in: Proc. of ICFEM 02, Vol. 2495 of LNCS, Springer, 2002, pp. 471–482.

[64] W. Krenn, B. K. Aichernig, Test case generation by contract mutation in Spec#, ENTCS 253 (2) (2009) 71–86.

[65] B. K. Aichernig, E. Jobstl, S. Tiran, Model-based mutation testing via symbolic refinement checking, Science of Computer Programming 97, Part 4 (0) (2015) 383 – 404, special Issue: Selected Papers from the 12th International Conference on Quality Software (QSIC 2012).

[66] B. K. Aichernig, Model-based mutation testing of reactive systems, in: Theories of Programming and Formal Methods, Springer, 2013, pp. 23–36.

[67] O. Alkrarha, J. Hassine, Muasmetal: An experimental mutation system for asmetal, in: 12th Int. Conf. on Information Technology - New Generations (ITNG), 2015, pp. 421–426.

[68] B. Aichernig, H. Brandl, E. Jobstl, W. Krenn, R. Schlick, S. Tiran, MoMut::UML model-based mutation testing for UML, in: Software Testing, Verification and Validation (ICST), 2015, pp. 1–8.

[69] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678.

[70] T. A. Budd, A. S. Gopal, Program testing by specification mutation, Comput. Lang. 10 (1) (1985) 63–73.

[71] P. E. Ammann, P. E. Black, W. Majurski, Using model checking to generate tests from specifications, in: 2nd Int. Conf. on Formal Engineering Methods, ICFEM '98, IEEE, 1998, pp. 46–54.

[72] W. Herzner, R. Schlick, H. Brandl, J. Wiessalla, Towards fault-based generation of test cases for dependable embedded software, Softwaretechnik-Trends 31 (3).

[73] A. Brillout, N. He, M. Mazzucchi, D. Kroening, M. Purandare, P. Rummer, G. Weissenbacher, Mutation-based test case generation for Simulink models, in: FMCO, 2009, pp. 208–227.

[74] M. Papadakis, N. Malevris, Searching and generating test inputs for mutation testing, SpringerPlus 2 (1) (2013) 1–12.

[75] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, Information and Software Technology 54 (9) (2012) 915–932.

[76] M. Papadakis, N. Malevris, Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing, Software Quality Journal 19 (4) (2011) 691–723.

[77] Z. Andrews, R. Payne, A. Romanovsky, A. Didier, A. Mota, Model-based development of fault tolerant systems of systems, in: Systems Conference (SysCon), 2013 IEEE International, 2013, pp. 356–363. doi:10.1109/SysCon.2013.6549906.

[78] Z. Andrews, J. Fitzgerald, R. Payne, A. Romanovsky, Fault modelling for systems of systems, in: Autonomous Decentralized Systems (ISADS), 2013 IEEE Eleventh International Symposium on, 2013, pp. 1–8. doi:10.1109/ISADS.2013.6513445.

Appendix A. *Circus* operational semantics

We reproduce here part of the *Circus* operational semantics as presented in [45].

As already said, as usual, the operational semantics of *Circus* is based on a transition relation that associates configurations. For processes, the configurations are processes themselves. For actions, they are triples as explained in Section 2.1.

To give the operational semantics of a process, we use a novel construct to define a process. It records the current local state using a constraint and a state assignment. The first transition rule for processes below introduces the record of the local state using a (list of) fresh symbolic variable(s) $\mathtt{w_0}$. The constraint defines that $\mathtt{w_0}$ is (are) of the appropriate type(s), and in the state assignment $\mathtt{w_0}$ is assigned to the state component(s) $\mathtt{x}$. In all rules, the symbolic variables introduced are assumed to be fresh.

$$\frac{}{\left(\begin{array}{l}\mathtt{begin}\\ \quad \mathtt{state}\,[\,\mathtt{x} : \mathtt{T}\,]\\ \quad \bullet\, \mathtt{A}\\ \mathtt{end}\end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l}\mathtt{begin}\\ \quad \mathtt{state}\,[\,\mathtt{x} : \mathtt{T}\,] \mid \mathtt{loc}\,(\mathtt{w_0} \in \mathtt{T} \mid \mathtt{x} := \mathtt{w_0})\\ \quad \bullet\, \mathtt{A}\\ \mathtt{end}\end{array}\right)} \tag{A.1}$$

The second transition rule for processes, which we omit here for conciseness, applies to the extended form of a basic process. The rule allows a process to evolve in accordance with the evolution of its main action in the state defined by the `loc` clause. We, therefore, focus in the sequel on the transition relation for actions.

The evolution of an output prefixing $\mathtt{d!e} \rightarrow \mathtt{A}$ is labelled. The label $\mathtt{d.w_0}$ involves the fresh constant $\mathtt{w_0}$; the new constraint defines its value to be that of $\mathtt{e}$ in the current state $\mathtt{s}$. The remaining action to be executed is $\mathtt{A}$.

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d!e} \rightarrow \mathtt{A}) \xrightarrow{\mathtt{d!w_0}} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{w_0} = \mathtt{e}) \mid \mathtt{s} \models \mathtt{A})} \tag{A.2}$$

The transition rule for an input prefixing $\mathtt{d?x} \rightarrow \mathtt{A}$ is as follows.

$$\frac{c \wedge T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x} : \mathtt{T} \rightarrow \mathtt{A}) \xrightarrow{\mathtt{d?w_0}} (\mathtt{c} \wedge \mathtt{w_0} \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\,x} := \mathtt{w_0} \models \mathtt{let\,x} \bullet \mathtt{A})} \tag{A.3}$$

The label is $\mathtt{d?w_0}$. In the new the state, $\mathtt{x}$ is declared and assigned $\mathtt{w_0}$. The only restriction on $\mathtt{w_0}$ is that it has the same type as $\mathtt{d}$. The remaining action $\mathtt{let\ x \bullet A}$ records the fact that $\mathtt{x}$ is in scope in $\mathtt{A}$ as a local variable. The construct $\mathtt{let\ x \bullet A}$ has been introduced specifically for use in the operational semantics. When $\mathtt{A}$ terminates, a rule for $\mathtt{let\ x \bullet Skip}$ closes the scope of $\mathtt{x}$ in the state and removes the $\mathtt{let\ x}$ declaration.

For an internal choice $\mathtt{A_1 \sqcap A_2}$, silent transitions are available to either $\mathtt{A_1}$ or $\mathtt{A_2}$ (in a configuration with the same constraint and state assignment).

$$\frac{c}{(\mathtt{c \mid s} \models \mathtt{A_1 \sqcap A_2}) \xrightarrow{\epsilon} (\mathtt{c \mid s} \models \mathtt{A_1})} \qquad \frac{c}{(\mathtt{c \mid s} \models \mathtt{A_1 \sqcap A_2}) \xrightarrow{\epsilon} (\mathtt{c \mid s} \models \mathtt{A_2})} \tag{A.4}$$

The treatment of parallelism is more subtle. We introduce a new form of action $\mathsf{par}\ s \mid x \bullet A$ to record a local state s of the parallel action A, with write control over the variables in x. The first transition rule for a parallelism defines a silent transition that rewrites it in terms of this new construct.

The rule below allows evolutions of the first parallel action $\mathtt{A_1}$ that are either silent or do not involve a channel in the synchronisation set to be reflected in the parallelism. A similar omitted rule considers independent evolutions of $\mathtt{A_2}$.

$$\frac{(\mathtt{c \mid s_1} \models \mathtt{A_1}) \xrightarrow{\mathtt{l}} (\mathtt{c_3 \mid s_3} \models \mathtt{A_3}) \quad \mathtt{l} = \epsilon \vee \mathrm{chan}\,\mathtt{l} \notin cs}{\left(\begin{array}{l}\mathtt{c \mid s}\\ \models\\ \left(\begin{array}{c}(\mathtt{par\ s_1 \mid x_1 \bullet A_1})\\ [\![\mathtt{cs}]\!]\\ (\mathtt{par\ s_2 \mid x_2 \bullet A_2})\end{array}\right)\end{array}\right) \xrightarrow{\mathtt{l}} \left(\begin{array}{l}\mathtt{c_3 \mid s}\\ \models\\ \left(\begin{array}{c}(\mathtt{par\ s_3 \mid x_1 \bullet A_3})\\ [\![\mathtt{cs}]\!]\\ (\mathtt{par\ s_2 \mid x_2 \bullet A_2})\end{array}\right)\end{array}\right)} \tag{A.5}$$

The next rule is for when the parallel actions can evolve by synchronising. In particular, $\mathtt{A_1}$ can carry out an input $\mathtt{d?w_1}$, and $\mathtt{A_2}$ an output $\mathtt{d!w_2}$, where $\mathtt{d}$ is a channel in the synchronisation set, and the values communicated are equal. The transition rule establishes that, in this case, the parallelism as a whole actually performs an output. The new constraint records the restriction that $\mathtt{w_1 = w_2}$.

$$\frac{\begin{array}{c}(\mathtt{c \mid s_1} \models \mathtt{A_1}) \xrightarrow{\mathtt{d?w_1}} (\mathtt{c_3 \mid s_3} \models \mathtt{A_3}) \qquad (\mathtt{c \mid s_2} \models \mathtt{A_2}) \xrightarrow{\mathtt{d!w_2}} (\mathtt{c_4 \mid s_4} \models \mathtt{A_4})\\ \mathtt{d} \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2\end{array}}{\left(\begin{array}{l}\mathtt{c \mid s}\\ \models\\ \left(\begin{array}{c}(\mathtt{par\ s_1 \mid x_1 \bullet A_1})\\ [\![\mathtt{cs}]\!]\\ (\mathtt{par\ s_2 \mid x_2 \bullet A_2})\end{array}\right)\end{array}\right) \xrightarrow{\mathtt{d!w_2}} \left(\begin{array}{l}\mathtt{c_3 \wedge c_4 \wedge w_1 = w_2 \mid s}\\ \models\\ \left(\begin{array}{c}(\mathtt{par\ s_3 \mid x_1 \bullet A_3})\\ [\![\mathtt{cs}]\!]\\ (\mathtt{par\ s_4 \mid x_2 \bullet A_4})\end{array}\right)\end{array}\right)} \tag{A.6}$$

Similar rules apply when $\mathtt{A_1}$ can output and $\mathtt{A_2}$ input, or when both $\mathtt{A_1}$ and $\mathtt{A_2}$ can output. When they can both input, the parallelism also performs an input.

Perhaps the most interesting rule is the one that applies when both parallel actions

have terminated. In this case, the parallelism terminates.

$$\frac{c}{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \left(\begin{array}{l} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{Skip}) \\ \quad [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{Skip}) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} (\mathtt{c} \mid (\exists\, \mathtt{x}_2' \bullet \mathtt{s}_1) \wedge (\exists\, \mathtt{x}_1' \bullet \mathtt{s}_2) \models \mathtt{Skip})} \tag{A.7}$$

The state after the parallelism is defined by composing the local states of the parallel actions. We keep from the local state $\mathtt{s}_1$ of the first action only the changes to the variables in its name set $\mathtt{x}_1$. This is achieved by hiding (quantifying) the final value of the variables in the complement set $\mathtt{x}_2$. The same applies to $\mathtt{s}_2$. The conjunction of the quantifications defines the new state. We observe that, alternatively, we can define the new state as $\mathtt{s}_1;\ \mathtt{end}\,\mathtt{x}_2 \wedge \mathtt{s}_2;\ \mathtt{end}\,\mathtt{x}_1$.

Rules for external choice require similar considerations. Actions in an external choice can evolve independently, with local access to all variables, until the choice is made, and consequently, the local changes become global. The new form of action $(\mathsf{loc}\ c \mid s \bullet A_1) \boxplus (\mathsf{loc}\ c \mid s \bullet A_2)$ records the initial state locally.

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_1 \,\Box\, \mathtt{A}_2) \xrightarrow{\epsilon} (\mathtt{c} \mid \mathtt{s} \models (\mathsf{loc}\ \mathtt{c} \mid \mathtt{s} \bullet \mathtt{A}_1) \boxplus (\mathsf{loc}\ \mathtt{c} \mid \mathtt{s} \bullet \mathtt{A}_2))} \tag{A.8}$$

Termination can resolve the choice.

$$\frac{c_1}{(\mathtt{c} \mid \mathtt{s} \models (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathbf{Skip}) \boxplus (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A})) \xrightarrow{\epsilon} (\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathbf{Skip})} \tag{A.9}$$

Since external choice is commutative, similar rules apply for each of the actions in the choice. We present just one of the two rules in each case. The next rule establishes that silent transitions do not resolve the choice.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\epsilon} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)}{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \left(\begin{array}{l} (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \\ \boxplus \\ (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \left(\begin{array}{l} (\mathsf{loc}\ \mathtt{c}_3 \mid \mathtt{s}_3 \bullet \mathtt{A}_3) \\ \boxplus \\ (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right)} \tag{A.10}$$

An event, however, does resolve the choice.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad \mathtt{l} \neq \epsilon}{(\mathtt{c} \mid \mathtt{s} \models (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \boxplus (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2)) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)} \tag{A.11}$$

For a hiding $\mathtt{A}_1 \setminus \mathtt{cs}$, the rules allow evolution of $\mathtt{A}_1$ to lead to evolution of the hiding itself. In the rule below, evolution does not involve a hidden channel, so the label for the hiding transition is that for the $\mathtt{A}_1$ transition.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \quad \text{chan } \mathtt{l} \notin cs}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \setminus \mathtt{cs}) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2 \setminus \mathtt{cs})} \tag{A.12}$$

If, on the other hand, $\mathtt{A}_1$ can communicate on a hidden channel, the corresponding evolution of the hiding is silent.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \quad \mathtt{l} = \epsilon \vee \text{chan } \mathtt{l} \in cs}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \setminus \mathtt{cs}) \xrightarrow{\epsilon} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2 \setminus \mathtt{cs})} \tag{A.13}$$

An omitted rule specifies that if $\mathtt{A}_1$ terminates, so does the hiding.

Appendix B. Emergency Response System

In this appendix, we include another example to illustrate the approach proposed in this paper. We consider the Emergency Response System (ERS), introduced in [77]. Targets, that is, incidents requiring emergency response, are identified by callers, that is, members of the public, using the ERS and a set of operationally independent subsystems, such as Phone System, Radio System, Call Center, and Emergency Response Unit (ERU). The ERS must ensure that every call should be sent to the correct target. More details about the ERS can be found in [78]. It is used in [52] to assess the deadlock detection of a prototype model checker for *Circus*.

The *Circus* specification in Figure B.20 models a subset of the ERU, focusing on the behavior of an emergency response unit manager, specified by the process ERU, and of a caller, process $InitiateRescueOrFault$. The latter sends rescue service requests to the ERU and can trigger a message-drop fault, to be detected and treated by a fault-recovery component, specified by a process $Recovery$ omitted here.

All three processes run concurrently and synchronise on the sets of channels indicated in the definition of their parallel composition. In broad terms, the ERU process manages the amount of available and allocated response units. $InitiateRescueOrFault$ asks for idle units and the ERU allocates them accordingly. If a fault is triggered, $Recovery$ logs the occurrence and resend the dropped message to the manager.

This example illustrates mutations that may affect concurrency. The operators $ppoParSeq$ and $ppoParInt$ are applicable in this context and yield interesting results that we discuss in the following sections.

Appendix B.1. Mutations by ppoParInt

The mutation operator $ppoParInt$ replaces a parallel composition with an interleaving, causing the concurrent execution to take place without synchronization between the processes. For this example, this operator can be applied in both parallel compositions to yield the mutants shown in Figure B.21.

Both mutants can be killed by test cases based on the fact that the minimal acceptance set for the original $ERSystem$ after the empty trace contains only the event $start\_rescue$. Other events available due to the mutation are forbidden continuations.

```
process ERU ≙ begin
  state Control == [ allocated, total_erus : ℕ ]
  InitControl == [ Control′ | allocated′ = 0 ∧ total_erus′ = 5 ]
  AllocateState == [ ΔControl | allocated′ = allocated + 1 ]
  Allocate ≙ allocate_idle_eru → AllocateState ; Choose
  ServiceState == [ ΔControl | allocated′ = allocated − 1 ]
  Service ≙ service_rescue → ServiceState ; Choose
  Choose ≙ if ( allocated = 0 ) ⟶ Allocate
           [] ( allocated = total_erus ) ⟶ Service
           [] ( allocated > 0 ∧ allocated < total_erus ) ⟶
                Allocate □ Service
           fi
  • InitControl; Choose
end

process InitiateRescueOrFault ≙ begin
  CallCentreStart ≙ start_rescue → FindIdleEru
  FindIdleEru ≙ find_idle_erus → (IdleEru □ (wait → FindIdleEru))
  IdleEru ≙ allocate_idle_eru → send_rescue_info_to_eru → IR1
  IR1 ≙ process_message → FAReceiveMessage □ fault_activation → IR2
  FAReceiveMessage ≙ receive_message → ServiceRescue
  ServiceRescue ≙ service_rescue → CallCentreStart
  IR2 ≙ IR2Out □ error_detection → FAStartRecovery
  IR2Out ≙ drop_message → target_not_attended → CallCentreStart
  FAStartRecovey ≙ start_recovery → end_recovery → ServiceRescue
  • CallCentreStart
end

channelset ERUSignals == {| allocate_idle_eru, service_rescue |}
channelset RecoverySignals == {| start_recovery, end_recovery |}
process ERSystem ≙
 (InitiateRescueOrFault [[ ERUSignals ]] ERU) [[ RecoverySignals ]] Recovery
```

Figure B.20: ERS Specification

```
process ppoParIntERSystem1 ≙
    (InitiateRescueOrFault ||| ERU) [[ RecoverySignals ]] Recovery

process ppoParIntERSystem2 ≙
    (InitiateRescueOrFault [[ ERUSignals ]] ERU) ||| Recovery
```

Figure B.21: *ppoSeqInt ERSystem* mutants

process $parSeqERSystem1 \mathrel{\widehat{=}}$
$(InitiateRescueOrFault;\ ERU) \llbracket\ RecoverySignals\ \rrbracket\ Recovery$

process $parSeqERSystem2 \mathrel{\widehat{=}}$
$(InitiateRescueOrFault \llbracket\ ERUSignals\ \rrbracket\ ERU);\ Recovery$

Figure B.22: Mutations by $ppoParSeq$

The test cases $T_T(s, a_1)$ and $T_F(s, X)$ below are based on the empty trace $s = \langle\rangle$, forbidden continuation $a_1 = start\_recovery$, and acceptance set $X = \{start\_rescue\}$.

$T_T(s, a_1) = pass \rightarrow start\_recovery \rightarrow fail \rightarrow$ **Stop**
$T_F(s, X) = fail \rightarrow start\_rescue \rightarrow pass \rightarrow$ **Stop**

They can both be used to kill $ppoParIntERSystem1$ and $ppoParIntERSystem2$.

Appendix B.2. Mutations by ppoParSeq

The parallel composition in the definition of $ERSystem$ is changed to a sequential composition by the concurrency mutation operator $ppoParSeq$. The result of the mutation is shown in Figure B.22. In the two produced mutants, the first process of the resulting sequential composition has a non-terminating looping behavior: both ERU and $InitiateRescueOrFault$ are non-terminating. This makes the second process in the sequential composition unreachable. As an strategy to kill the mutants, test cases can be based on traces that exercise events exclusively available in the unreachable process.

The behavior of the mutant $parSeqERSystem1$, for example, is restricted to that defined by $InitiateRescueOrFault$, at the left of the sequential composition, in parallel with $Recovery$. Such behavior is identical to that of $ERSystem$ while it holds that $allocated < total\_erus$ in the ERU process. To expose this, we need a trace to reach a state in $ERSystem$ where $allocated = total\_erus$ and check for a forbidden continuation $a_2 = allocate\_idle\_eru$ and acceptance set $X_2 = \{wait\}$. Below, we have a trace s_1 of both $parSeqERSystem1$ and $ERSystem$.

$$\begin{aligned} s_1 = \langle & start\_rescue, find\_idle\_eru, \\ & allocate\_idle\_eru, send\_rescue\_info\_to\_eru, \\ & fault\_activation, drop\_message, target\_not\_attended\rangle \end{aligned}$$

In $ERSystem$, in the execution of this trace, the state of the process ERU is changed by increasing in the value of $allocated$ by 1. Such trace can be repeatedly observed in the execution of $ERSystem$ up to five times, until $allocated = total\_erus$, when no more units are available and the minimal acceptance set is a single $wait$ event. The mutant $parSeqERSystem1$, on the other hand, is able to perform the forbidden continuation $allocate\_idle\_eru$ and, therefore, can be killed by the test cases below.

$T_T({s_1}^5 \frown \langle start\_rescue, find\_idle\_eru\rangle, allocate\_idle\_eru)$
$T_F({s_1}^5 \frown \langle start\_rescue, find\_idle\_eru\rangle, \{wait\})$

We use s_1^5 to represent the trace containing five consecutive copies of s_1. We omit the explicit definition of the above tests due to their size.

The mutation inflicted in $parSeqERSystem2$ removes the $Recovery$ process from the parallel execution, as it becomes the second part of the sequential composition. So, the events in this process are absent in the mutant. The following trace

$$s_2 = \langle start\_rescue, find\_idle\_eru, allocate\_idle\_eru, send\_rescue\_info\_to\_eru, \\ fault\_activation, error\_detection, start\_recovery \rangle$$

is both a trace of $parSeqERSystem2$ and $ERSystem$. In $ERSystem$, however, the next events are from $Recovery$, which is not reachable in the mutant $parSeqERSystem2$. So, the following tests can kill the mutant.

$$\begin{aligned} &T_T(s_2, end\_recovery) = \\ &\quad inc \rightarrow start\_rescue \rightarrow pass \rightarrow find\_idle\_eru \\ &\quad \rightarrow pass \rightarrow allocate\_idle\_eru \rightarrow pass \rightarrow send\_rescue\_info\_to\_eru \\ &\quad \rightarrow pass \rightarrow fault\_activation \rightarrow pass \rightarrow error\_detection \\ &\quad \rightarrow pass \rightarrow start\_recovery \rightarrow pass \rightarrow end\_recovery \\ &\quad \rightarrow fail \rightarrow \textbf{Stop} \end{aligned}$$

$$\begin{aligned} &T_F(s_2, \{log\_fault\}) = \\ &\quad inc \rightarrow start\_rescue \rightarrow fail \rightarrow find\_idle\_eru \\ &\quad \rightarrow fail \rightarrow allocate\_idle\_eru \rightarrow fail \rightarrow send\_rescue\_info\_to\_eru \\ &\quad \rightarrow fail \rightarrow fault\_activation \rightarrow fail \rightarrow error\_detection \\ &\quad \rightarrow fail \rightarrow start\_recovery \rightarrow fail \rightarrow log\_fault \\ &\quad \rightarrow pass \rightarrow \textbf{Stop} \end{aligned}$$

Considering the trace s_2, we can use the forbidden continuaton $end\_recovery$ and the minimal acceptance set $\{log\_fault\}$ to obtain the tests shown above.