Fault-Based Testing for Refinement in CSP

Ana Cavalcanti[1] and Adenilso Simao[2]

[1] University of York, UK – ana.cavalcanti@york.ac.br
[2] University of São Paulo, Brazil – adenilso@icmc.usp.br

Abstract. The process algebra CSP has been studied as a modeling notation for test derivation. Work has been developed using its trace and failure semantics, and their refinement notions as conformance relations. In this paper, we propose a procedure for online test generation for selection of finite test sets for traces refinement from CSP models, based on the notion of fault domains, that is, focusing on the set of faulty implementations of interest. We investigate scenarios where the verdict of a test campaign can be reached after a finite number of test executions. We illustrate the usage of the procedure with a small case study.

1 Introduction

Model-based testing (MBT) has received increasing attention due to its ability to improve productivity, by automating test planning, generation, and execution. The central artifact of an MBT technique is the model. It serves as an abstraction of the system under test (SUT), manageable by the testing engineers, and can be processed by tools to automatically derive tests. Most notations for test modeling are based on states; examples are Finite State Machines, Labelled Transition Systems, and Input/Output Transition Systems. Many test-generation techniques are available for them [7, 11, 25, 21]. Other notations use state-based machines as the underlying semantics [12, 16].

CSP [24] has been used as a modelling notation for test derivation. The pioneering work in [20] formalises a test-automation approach based on CSP. More recently, CSP and its model checker FDR [10] have been used to automate test generation with ioco as a conformance relation [19]. A theory for testing for refinement from CSP has been fully developed in [4].

Two sets of tests have been defined and proved to be exhaustive: they can detect any SUT that is non-conforming according to traces or failures refinement. Typically, however, these test sets are infinite. A few selection criteria have been explored: data-flow and synchronisation coverage [5], and mutation testing [1] for a state-rich version of CSP. The traditional approaches for test generation from state-based models have not been studied in this context.

Even though the operational semantics of CSP defines a Labelled Transition System (LTS), applying testing approaches based on states in this context is challenging: i) not every process has a finite LTS, and it is not trivial to determine when it has; ii) even if the LTS finite, in may not be deterministic; iii) for refinement, we are not interested in equivalence of LTS; and iv) to deal with failures, the notion of state needs to be very rich.

Here, we present a novel approach for selection of finite test sets from CSP models by identifying scenarios where the verdict of a campaign can be reached after a finite number of test executions. We adopt the concept of fault domain from state-based methods to constrain the possible faults in an SUT [22]. Fault-based testing is more general than the criteria above, since the test engineering can embed knowledge about the possible faults of the SUT into a fault domain to guide generation and execution [13]. We define a fault domain as a CSP process that is assumed to be refined by the SUT. With that, we establish that some tests are not useful, as they cannot reveal any new information about the SUT.

In addition, we propose a procedure for online generation of tests for traces refinement. Tests are derived and applied to the SUT and, based on the verdict, either the SUT is cast incorrect, or the fault domain is refined.

We present some scenarios where our procedure is guaranteed to provide a verdict after a finite number of steps. A simple scenario is that of a specification with a finite set of traces: unsurprisingly, after that set is exhaustively explored our procedure terminates. A more interesting scenario is when the SUT is incorrect; our procedure also always terminates in this case.

We have also investigated the scenario where the set of traces of the specification is infinite, but that of the SUT is finite and the SUT is correct. A challenge in establishing termination is that, while when testing using Mealy and finite state machines every trace of the model leads to a test, this is not the case with CSP. For example, for traces refinement, traces of the specification that lead to states in which all possible events are accepted give rise to no tests. After such a trace, the behavior of the SUT is unconstrained, and so does not need to be tested. Another challenge is that most CSP fault domains are infinite.

Our approach is similar to those adopted in the traditional finite state-machines setting, but addresses these challenges. We could, of course, change the notion of test and add tests for all traces. A test that cannot fail, however, is, strictly speaking, just a probe. For practical reasons, it is important to avoid such probes, which cannot really reveal faults.

The contributions of this paper are: 1) the introduction of the notion of fault domain in the context of a process algebra for refinement; 2) a procedure for online testing for traces refinement validated by a prototype implementation; and 3) the characterization of some scenarios in which the procedure terminates.

Next, in Section 2 we present background material: fault-based testing, and CSP and its testing theory. Section 3 casts the traditional concepts of fault-based testing in the context of CSP. Our procedure is presented in Section 4. Termination is studied in Section 5. Section 6 describes a prototype implementation of our procedure and its use in a case study. Finally, we conclude in Section 7.

2 Preliminaries

In this section, we describe the background material to our work.

2.1 CSP: testing and refinement

CSP is distinctive as a process algebra for refinement. In CSP models, systems and components are specified as reactive processes that communicate synchronously via channels. A prefixing $a \rightarrow P$ is a process that is ready to communicate by engaging in the event a and then behaves like P. The external choice operator $\Box$ combines processes to give a menu of options to the environment.

Example 1. The process *Counter* uses events *add* and *sub* to count up to 2.

$$Counter = add \rightarrow Counter1$$
$$Counter1 = add \rightarrow Counter2 \,\Box\, sub \rightarrow Counter$$
$$Counter2 = sub \rightarrow Counter1$$

$Counter1$ offers a choice to increase (*add*) or decrease (*sub*) the counter. □

Other operators combine processes in internal (nondeterministic) choice, parallel, sequence, and so on. Nondeterminism can also be introduced by interleaving and by hiding internal communications, for example.

There are three standard semantics for CSP: traces, failures, and failure-divergences, with refinement as the notion of conformance. As usual, the testing theory assumes that specifications and the SUT are free of divergence, which is observed as deadlock in a test. So, tests are for traces or failures refinement.

We write $P \sqsubseteq_T Q$ when P is trace-refined by Q; similarly, for $P \sqsubseteq_F Q$ and failures-refinement. In many cases, definitions and results hold for both forms of refinement, and we write simply $P \sqsubseteq Q$. In all cases, $P \sqsubseteq Q$ requires that the observed behaviours of Q (either its traces or failures) are all possible for P.

The CSP testing theory adopts two testability hypothesis. The first is often used to deal with a nondeterministic SUT: there is a number k such that, if we execute a test k times, the SUT produces all its possible behaviours. (In the literature, it appears in [15, 25, 14], for example, as fairness hypothesis or all-weather assumption.) The second testability hypothesis is that there is an (unknown) CSP process SUT that characterises the SUT.

The notion of execution of a test T is captured by a process $Execution^S_{SUT}(T)$ that composes the SUT and the test T in parallel, with all their common (specification) events made internal. Special events in T give the verdict: *pass*, *fail*, or *inc*, for inconclusive tests that cannot be executed to the end because the SUT does not have the trace that defines the test.

The testing theory also has a notion of successful testing experiment: a property $passes_{\sqsubseteq}(S, SUT, T)$ defines that the SUT passes the test T for specification S. A particular definition for $passes_{\sqsubseteq}(S, SUT, T)$ typically uses the definition of $Execution^S_{SUT}(T)$, but also explains how the information arising from it is used to achieve a verdict. For example, for traces refinement, we have the following.

$$passes_T(S, SUT, T) \mathrel{\widehat{=}} \forall\, t : traces\, [\![Execution^S_{SUT}(T)]\!] \bullet last(t) \neq fail$$

For a definition of $passes_{\sqsubseteq}(S, SUT, T)$ and a test suite TS, we use the notation $passes_{\sqsubseteq}(S, SUT, TS)$ as a shorthand for $\forall\, T : TS \bullet passes_{\sqsubseteq}(S, SUT, T)$.

In general, for a given definition of $passes_{\sqsubseteq}(S, SUT, T)$, we can characterise exhaustivity $Exhaust_{\sqsubseteq}(TS)$ of a test suite TS as follows.

Definition 1. *A test suite TS satisfies the property $Exhaust_{\sqsubseteq}(S, TS)$, that is, it is exhaustive for a specification S and a conformance relation $\sqsubseteq$ exactly when, for every process P, we have $S \sqsubseteq P \Leftrightarrow passes_{\sqsubseteq}(S, P, TS)$.*

Different forms of test give rise to different exhaustive sets. We use $Exhaust_{\sqsubseteq}(S)$ to refer to a particular exhaustive test suite for S and $\sqsubseteq$.

For a trace $\langle a_1, a_2, \dots \rangle$ with events a_1, a_2, and so on, and one of its forbidden continuations a, that is, an event a not allowed by the specification after the trace, the traces-refinement test $T_T(\langle a_1, a_2, \dots \rangle, a)$ is given by the process $inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow ... \rightarrow pass \rightarrow a \rightarrow fail$. In alternation, it gives an *inc* verdict and offers an event of the trace to the SUT, until all the trace is accepted, when it gives the verdict *pass*, but offers the forbidden continuation. It it is accepted, the verdict is *fail*. The exhaustive test set $Exhaust_T(S)$ for traces refinement includes all tests $T_T(t, a)$ formed in this way from the traces t and forbidden continuations a of the specification.

Example 2. We consider the specification $S_1 = a \rightarrow b \rightarrow S_1$. The exhaustive test set for for S_1 and traces refinement is sketched below.

$$\{\, pass \rightarrow b \rightarrow fail \rightarrow STOP, inc \rightarrow a \rightarrow pass \rightarrow a \rightarrow fail \rightarrow STOP,$$
$$inc \rightarrow a \rightarrow inc \rightarrow b \rightarrow pass \rightarrow b \rightarrow fail \rightarrow STOP, \dots\}$$

□

In [3], it is proven that $Exhaust_{\sqsubseteq_T}(S, Exhaust_T(S))$.

2.2 Fault-based testing

The testing activity is constrained by the amount of resources available. Some criteria is needed to select a finite subset of finite tests. Fault-based criteria consider that there is a fault domain, modelling the set of all possible faulty implementations [22, 13]. They restrict the set of required tests using the assumption that the SUT is in that domain [26]. Testing has to consider the possibility that the SUT can be any of those implementations, but no others [17].

For Finite State Machines (FSMs), many test-generation techniques assume that the SUT may have a combination of initialisation faults (that is, the SUT initialises in a wrong state), output faults (that is, the SUT produces a wrong output for a given input), transfer faults (that is, a transition of the SUT leads to the wrong states), and missing or extra states (that is, the set of states of the SUT is increased or decreased). Therefore, for a specification with n states, it is common that the fault domain is defined denotationally as "the set of FSMs (of a given class) with no more than m states, for some $m \geq n$." [9, 7, 11]. In this case, all faults above are considered, except for more extra states than $m - n$.

Fault domains can also be used to restrict testing to parts of the specification that the tester judges more relevant. For instance, some events of the specification can be trivial to implement and the tester may decide to ignore them. An

approach for modelling faults of interest, considering FSMs, is to assume that the SUT is a submachine of a given non-deterministic FSM, as in [13]. Thus, the parts of the SUT that are assumed to be correct can be easily modelled by a *copy* the specification; the faults are then modelled by adding extra transitions with the intended faults. Fault domains can also be modelled by explicitly enumerating the possible faulty implementations, known as mutants [8]. Thus, tests can be generated targetting each of those mutants, in turn.

In the next section, we define fault domains by refinement of a CSP process.

3 Fault-based testing in CSP

For CSP, we define a fault domain as a process $FD \sqsubseteq SUT$; it characterises the set of all processes that refine it. We use the term fault domain sometimes to refer to the CSP process itself and sometimes to the whole collection of processes it identifies. In the CSP testing theory, the specification and SUT are processes over the same alphabet of events. Accordingly, here, we assume that a fault domain FD uses only those events as well.

The usefulness of the concept of fault domain is illustrated below.

Example 3. For S_1 in Example 2, we first take just $FD_1 = RUN(\{a, b\})$ as a fault domain. For any alphabet A, the process $RUN(A)$ repeatedly offers all events in A. So, with FD_1, we add no extra information, since every process that uses only channels a and b trace refines FD_1. A more interesting example is $FD_2 = a \rightarrow (a \rightarrow FD_2 \ \Box \ b \rightarrow FD_2)$. In this case, the assumption that $FD_2 \sqsubseteq_T SUT$ allows us to eliminate the first and the third tests in Example 2, because an SUT that refines FD_2 always passes those tests. □

In examples, we use traces refinement as the conformance relation, and assume that we have a fixed notion of test. The concepts introduced here, however, are relevant for testing for either traces or failures refinement.

It is traditional in the context of Mealy machines to consider a fault domain characterised by the size of the machines, and so, finite. Here, however, if a fault domain FD has an infinite set of traces, it may have an infinite number of refinements. For traces refinement, for example, for each trace t, a process that performs just t refines FD. So, we do not assume that fault domains are finite.

Just like we define the notion of exhaustive test set to identify a collection of tests of interest, we define the notion of a complete test set, which contains the tests of interest relative to a fault domain.

Definition 2. *For a specification S, and a fault domain FD, we define a test set $TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S)$ to be complete, written $Complete^{S}_{\sqsubseteq}(TS, FD)$, with respect to FD if, and only if, for every implementation I in FD we have*

$$\neg\ (S \sqsubseteq I) \Rightarrow \exists\ T : TS \bullet \neg\ passes_{\sqsubseteq}(S, I, T)$$

This is a property based not on the whole of the fault domain, but just on its faulty implementations. For traces refinement, the exhaustive test set is given by $Exhaust_T(S)$ and the verdict by $passes_T(S, SUT, T)$ defined in Section 2.1.

If FD is the bottom of the refinement relation $\sqsubseteq$, then a complete test set TS is exhaustive. It is direct from Definition 2 the fact that a complete test set is a subset of the exhaustive test set and, therefore, unbiased, that is, it does not reject correct programs. We also need validity: only correct programs are accepted. This is also fairly direct as established in the theorem below.

Theorem 1 *Provided $FD \sqsubseteq SUT$, we have that*

$$\exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet complete(TS, FD) \wedge passes_{\sqsubseteq}(S, SUT, TS)$$

implies $S \sqsubseteq SUT$.

Proof

$$\exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet complete(TS, FD) \wedge passes_{\sqsubseteq}(S, SUT, TS)$$

$$= \exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet \quad \text{[Definition 2]}$$
$$(\neg\, (S \sqsubseteq SUT) \Rightarrow \exists\, T : TS \bullet \neg\, passes_{\sqsubseteq}(S, SUT, T)) \wedge$$
$$passes_{\sqsubseteq}(S, SUT, TS)$$

$$= \exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet \quad \text{[predicate calculus and definition of } passes\text{]}$$
$$(passes_{\sqsubseteq}(S, SUT, TS) \Rightarrow S \sqsubseteq SUT) \wedge passes_{\sqsubseteq}(S, SUT, TS)$$

$$\Rightarrow (S \sqsubseteq SUT) \quad \text{[predicate calculus]}$$

□

Finally, if an unbiased test is added to a complete set, the resulting set is still complete. Unbias follows from inclusion in the exhaustive test set.

An important set is those of the useless tests for implementations in the fault domain. The fact that we can eliminate such tests from any given test suite has an important practical consequence.

Definition 3.

$$Useless_{\sqsubseteq}(S, FD) = \{T : Exhaust_{\sqsubseteq}(S) \mid passes_{\sqsubseteq}(S, FD, T)\}$$

Since FD passes the tests in $Useless_{\sqsubseteq}(S, FD)$, all implementations in that fault domain also pass those tests, provided $passes_{\sqsubseteq}(S, P, T)$ is monotonic on P with respect to refinement. This is proved below.

Lemma 1. *For every I in FD, and for every $T : Useless_{\sqsubseteq}(S, FD)$, we have $passes_{\sqsubseteq}(S, I, T)$, if $passes_{\sqsubseteq}(S, P, T)$ is monotonic on P with respect to $\sqsubseteq$.*

Proof

$$FD \sqsubseteq I$$
$$\Rightarrow passes_{\sqsubseteq}(S, FD, T) \Rightarrow passes_{\sqsubseteq}(S, I, T) \quad \text{[monotonicity of } passes\text{]}$$
$$= T \in Useless_{\sqsubseteq}(S, FD) \Rightarrow passes_{\sqsubseteq}(S, I, T) \quad \text{[definition of } Useless_{\sqsubseteq}(S, FD)\text{]}$$

□

Example 4. We recall that the definition for $passes_T(S, SUT, T)$ is monotonic, as shown below, where we consider processes P_1 and P_2 such that $P_1 \sqsubseteq_T P_2$.

$$Execution^S_{P_1}(T) = (P_1 \,[\![\, \alpha S \,]\!]\, T) \backslash \alpha S \qquad \text{[definition of } Execution\text{]}$$

$$\Rightarrow Execution^S_{P_1}(T) \sqsubseteq_T (P_2 \,[\![\, \alpha S \,]\!]\, T) \backslash \alpha S$$
[monotonicity of CSP operators with respect to refinement]

$$= Execution^S_{P_1}(T) \sqsubseteq_T Execution^S_{P_2}(T) \qquad \text{[definition of } Execution\text{]}$$

$$\Rightarrow traces\,[\![Execution^S_{P_2}(T)]\!] \subseteq traces\,[\![Execution^S_{P_1}(T)]\!] \qquad \text{[definition of } \sqsubseteq_T\text{]}$$

$$\Rightarrow (\forall\, t : traces\,[\![Execution^S_{P_1}(T)]\!] \bullet last(t) \neq fail) \Rightarrow (\forall\, t : traces\,[\![Execution^S_{P_2}(T)]\!] \bullet last(t) \neq fail) \qquad \text{[predicate calculus]}$$

$$= passes_T(S, P_1, T) \Rightarrow passes_T(S, P_2, T) \qquad \text{[definition of } passes_T\text{]}$$

□

Typically, it is expected that the notions of $passes_{\sqsubseteq}(S, P, T)$ are monotonic on P with respect to the refinement relation $\sqsubseteq$: a testing experiment that accepts a process, also accepts its correct implementations.

It is important to note that there are tests that do become useless with a fault-domain assumption. This is illustrated below.

Example 5. In Example 3, the first and third tests of the exhaustive test set are useless as already indicated. For instance, we can show that FD_2 passes the first test $T_1 = pass \rightarrow b \rightarrow fail \rightarrow STOP$ as follows.

$$Execution^{S_1}_{FD_2}(T_1)$$

$$= (FD_2 \,[\![\, \{a, b\} \,]\!]\, T_1) \setminus \{a, b\} \qquad \text{[definition of } Execution\text{]}$$

$$= (pass \rightarrow (FD_2 \,[\![\, \{a, b\} \,]\!]\, b \rightarrow fail \rightarrow STOP)) \setminus \{a, b\}$$
[step law of parallelism]

$$= pass \rightarrow (FD_2 \,[\![\, \{a, b\} \,]\!]\, b \rightarrow fail \rightarrow STOP) \setminus \{a, b\} \qquad \text{[step law of hiding]}$$

$$= pass \rightarrow STOP \setminus \{a, b\} \qquad \text{[step law of parallelism]}$$

$$= pass \rightarrow STOP \qquad \text{[step law of hiding]}$$

So, $traces\,[\![Execution^{S_1}_{FD_2}(T_1)]\!] = \{\langle\rangle, \langle pass\rangle\}$, none of which finish with *fail*. □

4 Generating test sets

To develop algorithms to generate tests based on a fault domain, we need to consider particular notions of refinement, and the associated notions of test and verdict. In this paper, we present an algorithm for traces refinement.

A particular execution of the test can result in the verdicts *inc*, *pass* or *fail*. Due to nondeterminism in the SUT, the test may need to be executed multiple times, resulting in more than one verdict. We assume that the test is executed as many times as needed to observe all possible verdicts according to our testability hypothesis. So, for a test T and implementation SUT, we write $verd_{SUT}(T)$ to denote the set of verdicts observed when T is executed to test SUT.

If $fail \in verd_{SUT}(T)$, the SUT is faulty (if T is in $Exhaust_{\sqsubseteq}(TS)$). In this case, we can stop the testing activity, since the SUT needs to be corrected. Otherwise, we can determine additional properties of the SUT, considering the test verdicts. The SUT is a black box, but combining the knowledge that it is in the fault domain and has not failed a test, we can refine the fault domain.

If $fail \notin verd_{SUT}(T)$, both *inc* and *pass* bring relevant information. We consider a test $T_T(t, a)$, and recall that the SUT refines the fault domain FD. If $pass \in verd_I(T_T(t, a))$, then $t \in traces\,[\![SUT]\!]$, but $t \frown \langle a \rangle \notin traces\,[\![SUT]\!]$. Thus, the fault domain can be updated, since we have more knowledge about the SUT: it does not have the trace $t \frown \langle a \rangle$. Otherwise, if $verd_I(T_T(t, a)) = \{inc\}$, the trace t was not completely executed, and hence the SUT does not implement t. We can, therefore, update the fault domain as well.

In both cases, we include in the fault domain knowledge about traces not implemented. Information about implemented traces is not useful: given the definition of traces refinement, it cannot be used to reduce the fault domain.

Given a fault domain FD and a trace t, such that $t \notin traces\,[\![SUT]\!]$, we define a new fault domain as follows. First, we define a process $NOTTRACE(t)$, which tracks the execution of each event in t, behaving like the process $RUN(\Sigma)$ if the corresponding event of the trace does not happen. If we get to the end of t, then $NOTTRACE(t)$ prevents its last event from occurring. It, however, accepts any other event, and, at that point, also behaves like $RUN(\Sigma)$.

$$
\begin{aligned}
NOTTRACE(\langle a \rangle) &= \Box\, e : \Sigma \setminus \{a\} \bullet e \rightarrow RUN(\Sigma) \\
NOTTRACE(\langle a \rangle \frown t) &= a \rightarrow NOTTRACE(t) \\
&\quad \Box \\
&\quad (\Box\, e : \Sigma \setminus \{a\} \bullet e \rightarrow RUN(\Sigma))
\end{aligned}
$$

Formally, if the monitored trace is a singleton $\langle a \rangle$, then a is blocked by the process $NOTTRACE(\langle a \rangle)$. It offers in external choice all events except a: those in the set Σ of all events minus $\{a\}$. If a different event e happens, then $\langle a \rangle$ is no longer possible and the monitor accepts all events. If the monitored trace is $\langle a \rangle \frown t$, for a non-empty t, then, if a happens, we monitor t. If a different event e happens, then $\langle a \rangle \frown t$ is no longer possible and the monitor accepts all events.

We notice that $NOTTRACE$ is not defined for the empty trace, which is a trace of every process, and that, as required, $t \notin traces\,[\![NOTTRACE(t)]\!]$. On the other hand, for any trace s that does not have t as a prefix, we have that $s \in traces\,[\![NOTTRACE(t)]\!]$. To obtain a refined fault domain $FDU(t)$, we compose FD in parallel with $NOTTRACE(t)$.

$$FDU(t) = FD \,[\![\, \Sigma \,]\!]\, NOTTRACE(t)$$

```
1: procedure TestGen(S, FD_init, SUT)
2:     FD ← FD_init
3:     failed ← false
4:     TS ← ∅
5:     while ¬ (S ⊑_T FD) ∧ ¬ failed do
6:         Select a shortest t ∈ (traces ⟦FD⟧ ∩ traces ⟦S⟧) \ TS
7:         if initials(FD/t) \ initials(S/t) ≠ ∅ then
8:             Select a ∈ initials(FD/t) \ initials(S/t)
9:             verd ← ApplyTest(SUT, T_T(t, a))
10:            if fail ∈ verd then
11:                failed ← true
12:            else if pass ∈ verd then
13:                FD ← FDU(FD, t ⌢ ⟨a⟩)
14:            else                                   ▷ that is, verd = {inc}
15:                FD ← FDU(FD, t)
16:            end if
17:        else                          ▷ that is, initials(FD/t) \ initials(S/t) = ∅
18:            TS ← TS ∪ {t}
19:        end if
20:    end while
21:    return ¬ failed
22: end procedure
```

Fig. 1. Procedure for test generation

The parallelism requires synchronisation on all events and, therefore, controls the occurrence of events as defined by $NOTTRACE(t)$. So, the fault domain defined by $FDU(t)$ excludes processes that perform t.

Since $t \notin traces\,[\![SUT]\!]$ and $FD \sqsubseteq_T SUT$, then $FDU(t) \sqsubseteq_T SUT$. Thus, we have $FD \sqsubseteq_T FDU(t) \sqsubseteq_T SUT$. If the fault domain trace refines the specification S, we have that $S \sqsubseteq_T FD \sqsubseteq_T SUT$; thus, we can stop testing, since $S \sqsubseteq_T SUT$.

Based on these ideas, we now introduce a procedure *TestGen* for test generation. It is shown for a specification S, an implementation SUT, and an initial fault domain FD_{init}. In the case that there is no special information about the implementation, the initial fault domain can be simply $RUN(\Sigma)$.

TestGen uses local variables *failed*, to record whether a fault has been found as a result of a test whose execution gives rise to a *fail* verdict, and FD, to record the current fault domain. Initially, their values are **false** and FD_{init}. A variable TS records the set of traces for which tests are no longer needed, because all its forbidden continuations, if any, have already been used for testing.

The procedure loops until it is found that the specification is refined by the fault domain or a test fails. In each iteration, we select a trace t that belongs both to the specification and the fault domain (Step 6). A trace of the specification that is not of the fault domain is guaranteed to lead to an inconclusive verdict, as it is necessarily not implemented by the SUT.

Next, we check whether t has a continuation that is allowed by the fault domain FD, but is forbidden by S. If it has, we choose one of these forbidden

continuations a (Step 8). If not, t is not (or no longer) useful to construct tests, and is added to TS. A forbidden continuation a of S that is also forbidden by FD is guaranteed to be forbidden by the SUT. So, testing for a is useless.

The resulting test $T_T(t, a)$ is used and the set of verdicts *verd* is analysed as explained above, leading to an update of the fault domain. The value returned by the procedure indicates whether the SUT trace refines S or not.

Example 6. We consider as specification the *Counter* from Example 1. A few tests for traces refinement obtained by applying $T_T(t, a)$ to the traces t of *Counter* are sketched below in order of increasing length.

$T_T(\langle\,\rangle, sub) = pass \rightarrow sub \rightarrow fail \rightarrow STOP$
$T_T(\langle add, sub\rangle, sub) = inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow pass \rightarrow sub \rightarrow fail \rightarrow STOP$
$T_T(\langle add, add\rangle, add) = inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP$
$T_T(\langle add, add, sub, add\rangle, add) =$
$inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP$
$T_T(\langle add, add, sub, sub\rangle, sub) = \ldots$

This is, of course, an infinite set, arising from an infinite set of traces. We note, however, that there are no tests for a trace that has one more occurrence of *add* than *sub*, since, in such a state, *Counter* has no forbidden continuations.

The verdicts depend on the particular SUT; we consider below one example: $SUT = add \rightarrow add \rightarrow STOP$. We note that, at no point, we use our knowledge of the SUT to select tests. That knowledge is used just to identify the result of the tests used in our procedure.

In considering $TestGen(Counter, SUT, RUN(\Sigma))$, the first test we execute is $T_T(\langle\rangle, sub)$, whose verdict is *pass*. So, we have $\langle sub\rangle \notin traces\,[\![SUT]\!]$, and the updated fault domain is $FD_1 = NOTTRACE(\langle sub\rangle) = add \rightarrow RUN(\Sigma)$. The parallelism with the fault domain $RUN(\Sigma)$ does not change $NOTTRACE(\langle sub\rangle)$.

Counter is not refined by FD_1, which after the event *add* has arbitrary behaviour. The next test is $T_T(\langle add, sub\rangle, sub)$, whose verdict is *inc*. Thus, we have that $\langle add, sub\rangle \notin traces\,[\![SUT]\!]$. Now, the fault domain is FD_2 below.

$$
\begin{aligned}
&FD_2\\
&= FD_1\ [\![\,\Sigma\,]\!]\ NOTTRACE(\langle add, sub\rangle)\\
&= (add \rightarrow RUN(\Sigma))\ [\![\,\Sigma\,]\!]\ (sub \rightarrow RUN(\Sigma)\ \Box\ add \rightarrow add \rightarrow RUN(\Sigma))\\
&= add \rightarrow add \rightarrow RUN(\Sigma)
\end{aligned}
$$

The next test is $T_T(\langle add, add\rangle, add)$ with verdict *pass*. Thus, FD_3 is the process $add \rightarrow add \rightarrow sub \rightarrow RUN(\Sigma)$. Next, $T_T(\langle add, add, sub, add\rangle, add)$ gives verdict *inc*, and we get $FD_4 = add \rightarrow add \rightarrow sub \rightarrow sub \rightarrow RUN(\Sigma)$ when we update the fault domain. Finally, $T_T(\langle add, add, sub, sub\rangle, sub)$ has verdict *inc* as well. So, $FD_5 = add \rightarrow add \rightarrow sub \rightarrow STOP$ is the new domain. Since $Counter \sqsubseteq_T FD_5$, the procedure terminates indicating that SUT is correct. $\Box$

Our procedure, however, may never terminate. We discuss below some cases where we can prove that it does.

5 Generating test sets: termination

A specification that has a finite set of (finite) traces is a straightforward case, since it suffices to test with each trace and each forbidden continuation. Our procedure, however, can still be useful, because useless tests may be used if the fault domain is not considered. Our procedure can reduce the number of tests.

In this scenario, our procedure terminates because, for any maximal trace t of the specification (that is, a trace that is not a prefix of any other of its traces), all events are forbidden continuations. Thus, once t is selected and all tests derived from the forbidden continuations are applied, either we find a fault, or the fault domain is refined to a process that has no traces that extends t.

When all maximal traces of the specification are selected (and the corresponding tests are applied), if no test returns a *fail* verdict, no trace of the fault domain extends a maximal trace of the specification. Thus, if no test returns a *fail* verdict, any trace of the fault domain is a trace of the specification and the procedure stops indicating success, since, in this case, $S \sqsubseteq_T FD$.

We now discuss a scenario where the specification does not have a finite set of traces, but the SUT does. Once a trace t is selected, if the set of events, and, therefore, forbidden continuations is finite, with the derived tests, we can determine whether or not the SUT implements any of the forbidden continuations. Moreover, if no *pass* verdict is observed, t itself is not implemented.

We note that if the SUT is incorrect, that is, it does not trace refine the specification, the procedure always terminates.

Lemma 2. *If* $\neg\ (S \sqsubseteq_T SUT)$*, then* $TestGen(S, FD_{init}, SUT)$ *terminates (and returns false), for any fault domain* FD_{init} *and finite* SUT*.*

Proof By $\neg\ (S \sqsubseteq_T SUT)$, there exists a trace $s \in traces\,[\![SUT]\!] \setminus traces\,[\![S]\!]$. Let t be the longest prefix of s that is a trace of S, that is, the longest trace in $pref(s) \cap traces\,[\![S]\!]$, which gives rise to the shortest test that reveals an invalid prefix of s. Let a be such that $t \frown \langle a \rangle \in traces\,[\![SUT]\!] \setminus traces\,[\![S]\!]$. We know that a is a forbidden continuation of t, since $t \in traces\,[\![S]\!]$, but $t \frown \langle a \rangle \notin traces\,[\![S]\!]$. Moreover, since $traces\,[\![SUT]\!] \subseteq traces\,[\![FD]\!]$, it follows that $t \frown \langle a \rangle \in traces\,[\![FD]\!]$; hence $a \in initials(FD/t) \setminus initials(S/t)$. Thus, there exists a test $T_T(t, a)$ which, when applied to the SUT produces a *fail* verdict.

Since t is the longest trace in $pref(s) \cap traces\,[\![S]\!]$, tests generated for any prefix of t do not exclude t from the traces of the updated fault domain. Moreover, the event a remains in $initials(FD/t) \setminus initials(S/t)$, since no tests for traces longer than t are applied before t. Therefore, the test $T_T(t, a)$ is applied (unless a test for a trace of the same length of t is applied and the verdict is *fail*, in which case the result also follows). In this case, $TestGen(S, FD_{init}, SUT)$ assigns *true* to the variable *failed*, since the verdict is *fail* and terminates with $\neg$ *failed*, that is, *false*. □

Now we consider the case when the SUT is correct and finite. For some specifications, like the *Counter* from Example 1 the procedure terminates, but not for all specifications as illustrated below.

Example 7. We consider $UNBOUNDED = a \rightarrow UNBOUNDED \,\Box\, b \rightarrow STOP$, the initial fault domain $FD_{init} = RUN(\Sigma)$, where $\Sigma = \{a, b\}$, and the SUT $STOP$. In $TestGen(UNBOUNDED, SUT, RUN(\Sigma))$, the first trace we choose is $\langle\rangle$, for which there is no forbidden continuation, and so, no test. The next trace is $\langle a\rangle$, for which again there is no forbidden continuation. For $\langle b\rangle$, the events a and b are forbidden continuations; the test $T_T(\langle b\rangle, a)$ results in an *inc* verdict. Thus, the fault domain is updated to $FD_1 = FDU(FD_{init}, \langle b\rangle)$ below.

$$\begin{aligned} &FD_1 = FD_{init} \,[\![\, \Sigma \,]\!]\, NOTTRACE(\langle b\rangle) \\ &= RUN(\Sigma) \,[\![\, \Sigma \,]\!]\, a \rightarrow RUN(\Sigma) \\ &= a \rightarrow RUN(\Sigma) \end{aligned}$$

As expected, $\langle b\rangle$ is not a trace of the fault domain anymore and no further tests are generated for it: it is never again selected in Step 6.

The next trace we select is $\langle a, a\rangle$, for which there is no forbidden continuation. Then, we select $\langle a, b\rangle$, with forbidden continuations a and b. $T_T(\langle a, b\rangle, a)$ is executed with an *inc* verdict. The next fault domain is $FD_2 = FDU(FD_1, \langle a, b\rangle)$.

$$\begin{aligned} &FD_2 = FD_1 \,[\![\, \Sigma \,]\!]\, NOTTRACE(\langle a, b\rangle) \\ &= (a \rightarrow RUN(\Sigma)) \,[\![\, \Sigma \,]\!]\, (b \rightarrow RUN(\Sigma) \,\Box\, a \rightarrow a \rightarrow RUN(\Sigma)) \\ &= a \rightarrow a \rightarrow RUN(\Sigma) \end{aligned}$$

In fact, the refined fault domains are always of the form

$$a \rightarrow a \rightarrow ... \rightarrow a \rightarrow RUN(\Sigma)$$

This is because there is no test generated for a trace $\langle a\rangle^k$, for $k \geq 0$. So, the procedure does not terminate. This happens for any correct SUT with respect to the specification $UNBOUNDED$. For an incorrect SUT, the procedure terminates. □

Example 8. We now consider *Counter* and why our procedure stops for its correct finite implementations. First, we note that our procedure uses traces of increasing length for deriving and applying tests, and for a finite SUT, there is a k such that all traces of the SUT are shorter than k. We consider a trace $t \in traces\,[\![Counter]\!]$ of length k. There are three possibilities for $Counter/t$. If $Counter/t = Counter$ or $Counter/t = Counter2$, we have $initials(Counter/t) \neq \Sigma$ and, thus, there is a test $T_T(t, a)$ for a forbidden *sub* or *add*. The verdict for this test is *inc* because the SUT has no trace of the length of t and the fault domain is updated, removing t as a trace of the fault domain and, thus, as a possible trace of the SUT. If $Counter/t = Counter1$, we have that $initials(Counter/t) = \Sigma$ and no test can be derived from t. However, for each trace s, such that $t \frown s \in traces\,[\![Counter]\!]$, s starts with either *add* or *sub*. In either case, a test will be generated, since $Counter/t \frown \langle add\rangle = Counter2$ and $Counter/t \frown \langle sub\rangle = Counter1$, for which there are tests, as seen before. For those tests, the verdict is *inc*, the fault domain is similarly updated, and

the traces $t \frown \langle add \rangle$ and $t \frown \langle sub \rangle$ are removed. The fact that t for which $Counter/t = Counter1$ cannot be arbitrarily extended just to traces without tests is the key property required for the procedure to terminate.

For some specifications, like *UNBOUNDED*, there may be no tests for an unboundedly long trace. In this case, a correct SUT does not fail and, in spite of this, no test is applied that prunes the fault domain. □

To characterise the above termination scenario, we introduce some notation.

Given traces r and t, we say that r is a prefix of t, denoted $r \leq t$ if there exists s, such that $r \frown s = t$. A prefix is proper, denoted $r < t$, if $s \neq \langle\rangle$. We say that t is a (proper) suffix of r if, and only if, r is a (proper) prefix of t. We denote by $pref(t)$ all prefixes of t, that is, $pref(t) = \{s : \Sigma^* \mid s \leq t\}$, and by $ppref(t)$, all proper prefixes of t, that is, $ppref(t) = pref(t) \setminus \{t\}$. Similarly, we denote by $suff(t)$ the set of all suffixes of t.

For a process S and $k \geq 0$, we define the set $traces\,[\![S]\!]_k$ of the traces of S of length k. Formally, $traces\,[\![S]\!]_k = \{t : traces\,[\![S]\!] \mid \#t = k\}$. Another subset $hfc(S, FD)$ of traces of S includes those for which there is at least a forbidden continuation that takes into account the fault domain. Formally, $hfc(S, FD) = \{t : traces\,[\![S]\!] \mid initials(FD/t) \setminus initials(S/t) \neq \emptyset\}$. Importantly, for each $t \in hfc(S, FD)$, there exists at least one test $T_T(t, a)$ for a forbidden continuation a that is allowed by the fault domain. Finally, given a set of traces Q, we denote by $minimals(Q)$ the set of traces of Q that are not a proper prefix of another trace in Q. Formally, $minimals(Q) = \{t : Q \mid \neg\ \exists\, s : Q \bullet t < s\}$.

We use $hfc(S, FD)$ to define conditions for termination of the procedure.

Lemma 3. *For a specification S and a fault domain FD_{init}, if for any finite set of traces $P \subseteq traces\,[\![S]\!]$, there exists a $k \geq 0$, such that, for each $r \in traces\,[\![S]\!]_k$, we have that there is a prefix of r that is not in P and has a forbidden continuation, that is, $((pref(r) \setminus P) \cap hfc(S, FD_{init})) \neq \emptyset$, then $TestGen(S, FD_{init}, SUT)$ terminates for any finite SUT.*

Proof If $\neg\ (S \sqsubseteq_T SUT)$, by Lemma 2, the procedure terminates.

We, therefore, assume that $S \sqsubseteq_T SUT$, and so $traces\,[\![SUT]\!] \subseteq traces\,[\![S]\!]$. Finiteness of the SUT means that $traces\,[\![SUT]\!]$ is finite. Let $k \geq 0$ be such that, for each $r \in traces\,[\![S]\!]_k$, we have $((pref(r) \setminus traces\,[\![SUT]\!]) \cap hfc(S, FD_{init})) \neq \emptyset$. This k is larger than the size of the largest trace of SUT, since otherwise $pref(r) \setminus traces\,[\![SUT]\!]$ is empty, and it exists because $traces\,[\![SUT]\!]$ is finite.

Let now $Q = (pref(traces\,[\![S]\!]_k) \setminus traces\,[\![SUT]\!]) \cap hfc(S, FD_{init})$ and let $M = minimals(Q)$. Let $p \in traces\,[\![SUT]\!]$. Let $r \in traces\,[\![S]\!]_k$ be such that $p \leq r$. There is at least an $s \in pref(r)$, such that $p \leq s$ and $s \in hfc(S, FD_{init})$ because $((pref(r) \setminus traces\,[\![SUT]\!]) \cap hfc(S, FD_{init})) \neq \emptyset$. Without loss of generality, assume that s is the shortest such a trace. Thus, $s \in M$ and $p \in pref(M)$, since $p \leq s$. It follows that $traces\,[\![SUT]\!] \subseteq pref(M)$ since p is arbitrary.

For each $t \in M$, there exists $a \in initials(FD_{init}/t) \setminus initials(S/t)$, since $t \in hfc(S, FD_{init})$ and from the definition of $hfc(S, FD_{init})$. Then, if the test $T_T(t, a)$ is applied to the SUT, the verdict is inc, since $t \notin traces\,[\![SUT]\!]$ because

$t \in M \subseteq Q$ and $Q \cap \mathit{traces}\,[\![SUT]\!] = \emptyset$. In this case, the fault domain is updated so that t is not a trace of the fault domain anymore.

Thus, if all tests derived for each $t \in M$ are applied, we obtain a fault domain FD such hat $\mathit{traces}\,[\![FD]\!] \subseteq \mathit{pref}(M)$. As all traces in M have length at most k, eventually, all traces in M are selected (unless the procedure has already terminated) and the tests derived for those traces are applied. As $\mathit{pref}(M) \subseteq Q \subseteq \mathit{hfc}(S, FD_{init}) \subseteq \mathit{traces}\,[\![S]\!]$, it follows that $\mathit{traces}\,[\![FD]\!] \subseteq \mathit{traces}\,[\![S]\!]$, that is, $S \sqsubseteq_T SUT$. $\mathit{TestGen}(S, FD_{init}, SUT)$ then terminates, with $\mathit{failed} = \mathit{false}$, and the result is *true*.

□

One scenario where the conditions in Lemma 3 hold is if there is an event in the alphabet which is not used in the model. In this case, that event is always a forbidden continuation and thus a test is generated for all traces. Even though this can be rarely the case for a specification at hand, the alphabet can be augmented with a special event for the purpose, guaranteeing that the procedure terminates. Such an event would act as a *probe* event. As said before, in practice, it is best to avoid probes since the tests that they induce can reveal no faults.

6 Tool support and case studies

We have developed a prototype tool that implements our procedure. The tasks related to the manipulation of the CSP model, such as checking refinement, computing forbidden continuations, determining verdicts, and so on, are handled by FDR. The tool is implemented in Ruby. It submits queries (assert clauses) to FDR and parses FDR's results in order to perform the computations required by the procedure. Specifically, FDR is used in two points:

1. for checking whether the specification refines the fault domain (Line 5). It is a straightforward refinement check in FDR;
2. for computing initials and forbidden continuations (Lines 7 and 8). For instance, to compute the complement of $\mathit{initials}(S/t)$, we invoke FDR to check $S \sqsubseteq_T \mathit{TTHENANY}(t)$, where S is compared to the process $\mathit{TTHENANY}(t)$ that performs t and then any event e from Σ. It is defined as follows.

   $$\mathit{TTHENANY}(\langle\rangle) = \Box\, e : \Sigma \bullet e \rightarrow \mathit{STOP}$$
   $$\mathit{TTHENANY}(\langle a \rangle \frown t) = a \rightarrow \mathit{TTHENANY}(t)$$

   If t is a trace of S, counterexamples to this refinement check provide traces $t \frown \langle e \rangle$, where e is not in the set $\mathit{initials}(S/t)$. Thus, we obtain $\mathit{initials}(S/t)$ by considering the events in Σ for which no such counterexample exists.

Currently, our prototype calls FDR many times from scratch. As a future optimization, we will incorporate the caching of the internal results of the FDR, to speed up posterior invocations with the same model.

We have used our prototype to carry out two case studies, the Transputer-based sensor for autonomous vehicles in [23], and the Emergency Response System (ERS) in [2]. The sensor is part of an architecture where each sensor is

associated with a Transputer for local processing and can be part of a network of sensors. The ERS allows members of the public to identify incidents requiring emergency response; it is a system of operationally independent systems (a Phone System, a Call Center, an Emergency Response Unit, and so on). The ERS ensures that every call is sent to the correct target. It is used in [18] to assess the deadlock detection of a prototype model checker for *Circus*.

For each case study, we have randomly generated 1000 finite SUTs with the same event alphabet. The experimental results confirm what we expected from the lemmas of the previous section. Namely, all incorrect SUTs are identified and the procedure terminates for all finite SUTs identified in Lemma 3. The prototype tool and the CSP model for the sensor, the ERS and other examples are in `http://www.github.com/adenilso/CSP-FD-TGen`. The data for the SUTs used in this case study is also available.

7 Conclusions

In this paper, we have investigated how fault domains can be used to guide test generation from CSP models. We have cast core notions of fault-domain testing in the context of the CSP testing theory. For testing for traces refinement, we have presented a procedure which, given a specification and a fault domain, it tests whether an SUT trace refines the fault domain. If the SUT is incorrect, the procedure selects a test that can reveal the fault. In the case of a correct SUT, we have stated conditions that guarantee that the procedure terminates.

There are specifications for which the procedure does not terminate. We postulate that for those specifications, there is no finite set of tests that is able to demonstrate the correctness of the SUT. Finiteness requires extra assumptions about the SUT. We plan to investigate this point further in future work.

The CSP testing theory also includes tests for *conf*, a conformance relation that deals with forbidden deadlocks; together, tests for *conf* and traces refinement can be used to establish failures refinement. Another interesting failures-based conformance relation for testing from CSP models takes into account the asymmetry of controllability of inputs and outputs in the interaction with the SUT [6]. It is worth investigating how fault domains can be used to generate finite test sets for these notions of conformance.

Acknowledgements

The authors would like to thank the partial financial support of the following entities: Royal Society (Grant: NI150186), FAPESP (Grant: 2013/07375-0). The authors also are thankful to Marie-Claude Gaudel, for the useful discussion in an early version of this paper.

References

1. A. Alberto, A. L. C. Cavalcanti, M.-C. Gaudel, and A. Simao. Formal mutation testing for *Circus*. *IST*, 81:131–153, 2017.

2. Z. Andrews et al. Model-based development of fault tolerant systems of systems. In *SysCon*, pages 356–363, April 2013.
3. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *9th ICFEM*, volume 4789 of *LNCS*, pages 151–170. Springer, 2007.
4. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in *Circus*. *Acta Informatica*, 48(2):97–147, 2011.
5. A. L. C. Cavalcanti and M.-C. Gaudel. Data Flow coverage for *Circus*-based testing. In *FASE*, volume 8441 of *LNCS*, pages 415–429, 2014.
6. A. L. C. Cavalcanti and R. Hierons. Testing with Inputs and Outputs in CSP. In *FASE*, pages, 359-374, 2013.
7. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
8. KA El-Fakih et al. FSM-based testing from user defined faults adapted to incremental and mutation testing. *Programming and Computer Software*, 38(4):201–209, 2012.
9. S. Fujiwara and G. von Bochmann. Testing non-deterministic state machines with fault coverage. In *FORTE*, pages 267–280. North-Holland, 1991.
10. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 A Modern Refinement Checker for CSP. In *TACAS*, pages 187–201, 2014.
11. R. M. Hierons and H. Ural. Optimizing the length of checking sequences. *IEEE TC*, 55(5):618–629, 2006.
12. W.-L Huang and J. Peleska. Exhaustive model-based equivalence class testing. In *ICTSS*, pages 49–64, 2013.
13. I. Koufareva, A. Petrenko and N. Yevtushenko Test Generation Driven by User-defined Fault Models. In TestCom, pages 215–236, 1999.
14. G. Luo et al. Test selection based on communicating nondeterministic finite-state machines using a generalized Wp-method. *IEEE TSE*, 20(2):149–162, 1994.
15. A. J. R. G. Milner. *A Calculus of Communicating Systems*, volume 92. Springer Verlag, 1980.
16. A. Moraes et al. A family of test selection criteria for timed input-output symbolic transition system models. *SCP*, 126:52–72, 2016.
17. L. J. Morell. A theory of fault-based testing. *IEEE TSEg*, 16(8):844–857, Aug 1990.
18. A. Mota et al. Rapid prototyping of a semantically well founded Circus model checker. In *SEFM*, volume 8702 of *LNCS*, pages 235–249. Springer, 2014.
19. S. Nogueira, A. C. A. Sampaio, and A. C. Mota. Test generation from state based use case models. *FACJ*, 26(3):441–490, 2014.
20. J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. In *FM*, volume 1051 of *LNCS*, 1996.
21. A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE TC*, 54(9), 2005.
22. A. Petrenko et al. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.
23. P. J. Probert, D. Djian, and H. Hu. Transputer architectures for sensing in a robot controller: Formal methods for design. *Concurrency: Practice and Experience*, 3(4):283–292, 1991.
24. A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2011.
25. J. Tretmans. Test generation with inputs, outputs, and quiescence. In *TACAS'96*, volume 1055 of *LNCS*, pages 127–146. Springer, 1996.
26. Y. T. Yu and M. F. Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2):179–202, 2012.