Noname manuscript No.
(will be inserted by the editor)

Fault-Based Refinement-Testing for CSP

Ana Cavalcanti · Adenilso Simao

the date of receipt and acceptance should be inserted later

Abstract The process algebra CSP has been studied as a notation for model-based testing. Theoretical and practical work has been developed using its trace and failure semantics, and their refinement notions as conformance relations. Two sets of tests have been defined and proved to be exhaustive, in the sense that they can identify any SUT that is non-conforming with respect to the relevant refinement relation. However, these sets are usually infinite, and in this case, it is obviously not possible to apply them to verify the conformity of an SUT. Some classical selection criteria based on models have been studied. In this paper, we propose a procedure for online test generation for selection of finite test sets for traces refinement from CSP models. It is based on the notion of fault domains, 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 some case studies.

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 a model. It serves as an abstraction of the system under test (SUT), manageable by the testing engineers, and can be processed by tools to derive tests automatically.

Most modelling notations for testing 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 [11,17,35,28]. Other notations use state-based machines as the underlying semantics [18,22].

CSP [32] is a process algebra for refinement. It has been around for decades and availability of good tools has ensured adoption in industry. CSP has a denotational, an algebraic, and an operational semantics, which have been proved to be consistent. CSP also has extensions to deal with time [33], and has been combined with data-modelling notations to define state-rich process algebra [34,31,8].

University of York, UK – ana.cavalcanti@york.ac.uk
· University of São Paulo, Brazil – adenilso@icmc.usp.br

More recently, CSP has also been used as a modelling notation for test derivation. The seminal work in [27] formalises a test-automation approach based on CSP. More recently, CSP and its model checker FDR [15] have been used to automate test generation with **ioco** as a conformance relation [26]. A theory for testing for refinement from CSP has been fully developed in [4].

In [4], two sets of tests have been defined and proved to be exhaustive: they can identify any SUT that is non-conforming according to traces or failures refinement. Typically, however, these test sets are infinite, rendering them impractical for real applications. A few selection criteria have been explored: data-flow and synchronisation coverage [5], and mutation testing [1] for a state-rich version of CSP. As far as we know, 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 is finite, it may not be deterministic; (iii) for refinement, we are not interested in equivalence of LTS as in state-based approaches; 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 [29].

Fault-based testing is more general than the testing approaches induced by the selection criteria previously considered for CSP mentioned above. In fault-based testing, the test engineering can embed knowledge about the possible faults of the SUT into a fault domain to guide generation and execution [19].

Here, 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. In general terms, tests are derived using a CSP specification and a fault model, and applied to the SUT. Based on the verdict, either the SUT is cast incorrect, or the fault domain is refined and the procedure iterates to derive and apply further tests.

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 present a formal proof of the correctness of the procedure in these cases using a refinement calculus [24, 10]. We show that it refines a simple specification for an algorithm that gives a correct verdict. The invariant and variant that we use give insight into the design of the procedure. Moreover, the proof establishes that, even if the specification has an infinite set of traces and the SUT is correct, and so the procedure may not terminate, we have partial correctness. This means that, if the procedure does terminate, it does give the correct verdict.

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 in this case 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 but add to the cost of the testing activity.

In summary, the problem we address here is generation of a finite number of tests based on a CSP model, taking advantage of a fault model that captures partial information about the SUT. For that, we cast the core concepts of fault-based testing in CSP, and solve the problem for tests for traces refinement.

The contributions of this paper are as follows.

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;
3. Formal proof of correctness of the procedure;
4. Characterization of some scenarios in which the procedure terminates;
5. A number of case studies that show the practical relevance of the procedure.

Preliminary results have been published in [9]. Compared to that work, besides explanations throughout, we add the contributions 3, and 4 above.

The practical use of our procedure requires the definition of a fault-domain in CSP, and this may prove to be a hurdle in terms of the learning curve that it imposes. On the other hand, we do not require any knowledge or assumption about a state-based model of the SUT. An obvious limitation, however, is that the procedure does not terminate in all cases as explained above. We believe that in such cases there may well be no finite set of tests that can demonstrate correctness of the SUT. In practice, as usual, we need to consider another selection criteria. It can be as simple as a restriction on the length of traces used to define tests. Such restrictions may also be useful when the set of tests is finite, but very large.

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. Correctness and termination are studied in Section 5. Section 6 describes a prototype implementation of our procedure and our experiments. Finally, we conclude in Section 7, where we also present related and future work.

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. These are black boxes that interact with each other and with their environment via atomic, instantaneous, and synchronous events. A CSP model specifies processes by defining their patterns of interaction. A communication takes place via a channel, and an event may represent an input, output, or simple synchronisation on a channel.

We describe here core operators of CSP. A prefixing $a \to P$ is a process that is ready to communicate by engaging in the event a and then behaves like the process P. The external choice operator $\Box$ combines processes to give a menu of options to the environment. The following example illustrates the use of these operators.

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

$$\begin{aligned}
&\mathit{Counter} = \mathit{add} \to \mathit{Counter}1\\
&\mathit{Counter}1 = \mathit{add} \to \mathit{Counter}2 \;\Box\; \mathit{sub} \to \mathit{Counter}\\
&\mathit{Counter}2 = \mathit{sub} \to \mathit{Counter}1
\end{aligned}$$

*Counter*1 offers a choice to increase (event *add*) or decrease (event *sub*) the counter. In this case *add* and *sub* are channels that are used just for synchronisation; they do not communicate values. $\Box$

Other operators combine processes in internal (nondeterministic) choice ($\sqcap$), parallel ($[\![\ldots]\!]$), sequence ($;$), and so on. Nondeterminism can also be introduced by interleaving ($|||$), a form of parallelism in which the parallel processes do not communicate, and by hiding internal communications, for example. We say more about the operators in the sequel when they are used.

There are three standard semantics for CSP: traces, failures, and failures-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 [21,35,20], for example, as fairness hypothesis or all-weather assumption. The second hypothesis is that there is an (unknown) CSP process *SUT* that models the SUT. Thus, similarly to other MBT approaches, we assume that the specification and an SUT are modelled using the same notation, CSP.

The notion of execution of a test T for a specification S is captured by a CSP process $\mathit{Execution}^S_{SUT}(T)$ defined below. This notion is independent of the conformance relation and the kind of test that is used. In the definition we take advantage of the fact that the test T is also a process. The set αS contains the specification events, which are used by T and the SUT.

$$\mathit{Execution}^S_{SUT}(T) = (SUT \;[\![\; \alpha S \;]\!]\; T) \setminus \alpha S$$

The process $\mathit{Execution}^S_{SUT}(T)$ composes the SUT and the test T in parallel with αS as a synchronisation set. So, the parallelism requires that SUT and T synchronise on these events, but T can proceed independently when raising special events that give the verdict. These are *pass*, *fail*, or *inc*, respectively, for a sucessful execution of the test, for tests that fail, and for inconclusive tests that cannot be executed to the end because the SUT does not have the trace that defines the test. The events of the specification are hidden (using the CSP operator $\backslash \alpha S$), so that, in a test execution we can only observe the verdict events.

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 a 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 process P, the set $traces\,[\![P]\!]$ contains all traces of P, and for any trace t, its last element is given by $last(t)$. For simplicity, if t is the empty trace, $last(t) \neq fail$ is deemed to hold trivially. For a definition of $passes_{\sqsubseteq}(S, SUT, T)$ and a test suite TS, we use $passes_{\sqsubseteq}(S, SUT, TS)$ 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 $\langle a_1, a_2, \ldots \rangle$ with events a_1, $a_2, \ldots$, and one of its forbidden continuations a, that is, an event a not allowed by the specification after the trace $\langle a_1, a_2, \ldots \rangle$, the traces-refinement test $T_T(\langle a_1, a_2, \ldots \rangle, a)$ is given by the process $inc \rightarrow a_1 \rightarrow inc \rightarrow a_2 \rightarrow ... \rightarrow pass \rightarrow a \rightarrow fail$. In alternation, $T_T(\langle a_1, a_2, \ldots \rangle, a)$ 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. If 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 S.

Example 2 We consider the specification from Example 1. The exhaustive test set for *Counter* and traces refinement is sketched below.

$$\begin{array}{l} \{\, pass \rightarrow sub \rightarrow fail \rightarrow STOP, \\ \;\; inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP, \\ \;\; inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow pass \rightarrow sub \rightarrow fail \rightarrow STOP, \\ \;\; \ldots \} \end{array}$$

We note that there is no test for the trace $\langle add \rangle$, since after this event both *add* and *sub* are accepted, and, thus,there is no forbidden continuation. □

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. There usually is an infinite number of possible tests and it is obviously not feasible to apply them all. 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 [29, 19]. They restrict the set of required tests using the assumption that the SUT is in that domain [38]. Testing has to consider the possibility that the SUT can be any of those implementations, but can discard all other possibilities. It is in this way that a fault domain allows us to reduce the number of tests [23]. We note that, in spite of its name, the specification S itself and any number of its correct implementations may be in the fault domain.

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 state), and missing or extra states (that is, the set of states of the SUT is increased or decreased) [11]. 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 $m \geq n$." [14, 11, 17]. 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. In this case, an approach for modelling faults of interest, using FSMs, is to assume that the SUT is a submachine of a given nondeterministic FSM, as in [19]. Thus, the parts of the SUT that are assumed to be correct are modelled by a *copy* the specification, and the faults are 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 [13]. In these approaches, tests can be generated targeting 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, indistinctively, if the context makes it clear what we mean.

In the CSP testing theory, the specification and SUT are processes over the same alphabet of events. Otherwise, refinement is not meaningful. 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 We consider the specification $S_1 = a \to b \to S_1$. The infinite exhaustive test set for S_1 and traces refinement is sketched below.

$$\begin{aligned}&\{\, pass \to b \to fail \to STOP,\\ &\ \ inc \to a \to pass \to a \to fail \to STOP,\\ &\ \ inc \to a \to inc \to b \to pass \to b \to fail \to STOP,\\ &\ \ \ldots\}\end{aligned}$$

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 $FD_2 = a \to (a \to FD_2 \Box b \to FD_2)$. In this case, the assumption that $FD_2 \sqsubseteq_T SUT$ allows us to eliminate the first and the third tests in the set above, because an SUT that refines FD_2 always passes those tests. $\Box$

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. Thus, 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.

The following result relates the exhaustive and complete test sets.

Theorem 1 *If FD is the bottom of the refinement relation $\sqsubseteq$, then a complete test set TS is also exhaustive.*

Proof We prove two results arising from $Complete(TS, \bot)$, where we use $\bot$ to represent whatever is the least refined process, that is, the bottom of the order $\sqsubseteq$, and consider an arbitrary process I in the fault domain.

$$\begin{aligned}&Complete(TS, \bot)\\ &\Leftrightarrow \bot \sqsubseteq I \Rightarrow (\neg\,(S \sqsubseteq I) \Rightarrow \exists\, T : TS \bullet \neg\, passes_{\sqsubseteq}(S, I, T))\\ &\hspace{20em}[\text{definition of } Complete(TS, \bot)]\\ &= \neg\,(S \sqsubseteq I) \Rightarrow \exists\, T : TS \bullet \neg\, passes_{\sqsubseteq}(S, I, T) \qquad [\bot \sqsubseteq I \text{ holds for all } I]\\ &= (\forall\, T : TS \bullet passes_{\sqsubseteq}(S, I, T)) \Rightarrow S \sqsubseteq I \qquad [\text{predicate calculus}]\\ &= passes_{\sqsubseteq}(S, I, TS) \Rightarrow S \sqsubseteq I \qquad [\text{definition of } passes_{\sqsubseteq}(S, I, TS)]\end{aligned}$$

In addition, we have the following implication.

$$\begin{array}{ll}
Complete(TS, \bot) & \\
\Rightarrow TS \subseteq Exhaust_{\sqsubseteq}(S) & [\text{type of } TS \text{ in the definition of } Complete(TS, \bot)] \\
= \forall\, T : TS \bullet T \in Exhaust_{\sqsubseteq}(S) & [\text{definition of } \subseteq] \\
\Rightarrow \forall\, T : TS \bullet S \sqsubseteq I \Rightarrow passes_{\sqsubseteq}(S, I, T) & [\text{definition of } Exhaust_{\sqsubseteq}(S)] \\
= S \sqsubseteq I \Rightarrow \forall\, T : TS \bullet passes_{\sqsubseteq}(S, I, T) & [\text{predicate calculus}] \\
= S \sqsubseteq I \Rightarrow passes_{\sqsubseteq}(S, I, TS) & [\text{definition of } passes_{\sqsubseteq}(S, I, TS)]
\end{array}$$

So, we have that $Complete(TS, \bot) \Rightarrow (S \sqsubseteq I \Leftrightarrow passes_{\sqsubseteq}(S, I, TS))$. Therefore, $Complete(TS, \bot) \Rightarrow Exhaust_{\sqsubseteq}(TS)$. □

It is direct from Definition 2 that a complete test set is a subset of the exhaustive test set, and so unbiased: it does not reject a correct SUT. We also need validity: only a correct SUT is accepted. This is also fairly direct as shown below.

Theorem 2 *Provided* $FD \sqsubseteq SUT$, *we have that*

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

implies $S \sqsubseteq SUT$.

Proof

$$\begin{array}{ll}
\exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet complete(TS, FD) \land passes_{\sqsubseteq}(S, SUT, TS) & \\
= \exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet & [\text{Definition 2}] \\
\quad (\neg\, (S \sqsubseteq SUT) \Rightarrow \exists\, T : TS \bullet \neg\, passes_{\sqsubseteq}(S, SUT, T)) \land & \\
\quad passes_{\sqsubseteq}(S, SUT, TS) & \\
= \exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet & [\text{predicate calculus}] \\
\quad ((\forall\, T : TS \bullet passes_{\sqsubseteq}(S, SUT, T)) \Rightarrow (S \sqsubseteq SUT)) \land & \\
\quad passes_{\sqsubseteq}(S, SUT, TS) & \\
= \exists\, TS : \mathbb{P}\, Exhaust_{\sqsubseteq}(S) \bullet & [\text{definition of } passes_{\sqsubseteq}(S, SUT, TS)] \\
\quad (passes_{\sqsubseteq}(S, SUT, TS) \Rightarrow S \sqsubseteq SUT) \land passes_{\sqsubseteq}(S, SUT, TS) & \\
\Rightarrow (S \sqsubseteq SUT) & [\text{predicate calculus}]
\end{array}$$

□

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.

Theorem 3 *For any unbiased test* T, *we have the following property.*

$$Complete(TS, FD) \Rightarrow Complete(TS \cup \{T\}, FD)$$

Proof

$$\begin{array}{ll}
Complete(TS, FD) & \\
\Rightarrow \neg\, (S \sqsubseteq I) \Rightarrow \exists\, T : TS \bullet \neg\, passes_{\sqsubseteq}(S, I, T) & [\text{Definition 2}] \\
\Rightarrow \neg\, (S \sqsubseteq I) \Rightarrow \exists\, T : TS \cup \{T\} \bullet \neg\, passes_{\sqsubseteq}(S, I, T) & [\text{predicate calculus}]
\end{array}$$

$= \mathit{Complete}(TS \cup \{T\}, FD)$ [T is sound and definition of soundness]

$\square$

Similarly to the exhaustive test set, typically, a complete test set is infinite. Therefore, as already said, a practical technique still needs to use extra assumptions.

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 that we explore later.

Definition 3

$$\mathit{Useless}_{\sqsubseteq}(S, FD) = \{T : \mathit{Exhaust}_{\sqsubseteq}(S) \mid \mathit{passes}_{\sqsubseteq}(S, FD, T)\}$$

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

Theorem 4 *For every I in FD, and $T : \mathit{Useless}_{\sqsubseteq}(S, FD)$, we have $\mathit{passes}_{\sqsubseteq}(S, I, T)$, if $\mathit{passes}_{\sqsubseteq}(S, P, T)$ is monotonic on P with respect to $\sqsubseteq$.*

Proof

$FD \sqsubseteq I$

$\Rightarrow \mathit{passes}_{\sqsubseteq}(S, FD, T) \Rightarrow \mathit{passes}_{\sqsubseteq}(S, I, T)$ [monotonicity of *passes*]

$= T \in \mathit{Useless}_{\sqsubseteq}(S, FD) \Rightarrow \mathit{passes}_{\sqsubseteq}(S, I, T)$ [definition of $\mathit{Useless}_{\sqsubseteq}(S, FD)$]

$\square$

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

$\mathit{Execution}^S_{P_1}(T) = (P_1 \mathbin{[\![\, \alpha S \,]\!]} T) \setminus \alpha S$ [definition of $\mathit{Execution}^S_{P_1}(T)$]

$\Rightarrow \mathit{Execution}^S_{P_1}(T) \sqsubseteq_T (P_2 \mathbin{[\![\, \alpha S \,]\!]} T) \setminus \alpha S$

[monotonicity of CSP operators with respect to refinement]

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

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

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

[$\forall t : \mathit{traces}\, [\![\mathit{Execution}^S_{P_1}(T)]\!] \bullet t \in \mathit{traces}\, [\![\mathit{Execution}^S_{P_1}(T)]\!]$]

$= \mathit{passes}_T(S, P_1, T) \Rightarrow \mathit{passes}_T(S, P_2, T)$ [definition of $\mathit{passes}_T(S, P, T)$]

$\square$

Typically, it is expected that the notions of $\mathit{passes}_{\sqsubseteq}(S, P, T)$ are monotonic on P with respect to the refinement relation $\sqsubseteq$, so that 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 $T_1 = pass \to b \to fail \to STOP$ as follows.

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

$$= (FD_2 \,[\![\{a,b\}]\!]\, T_1) \setminus \{a,b\} \qquad \text{[definition of } Execution^{S_1}_{FD_2}(T_1)\text{]}$$

$$= (pass \to (FD_2 \,[\![\{a,b\}]\!]\, b \to fail \to STOP)) \setminus \{a,b\}$$
$$\text{[definitions of } FD_2 \text{ and } T_1\text{, and step law of parallelism]}$$

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

$$= pass \to STOP \setminus \{a,b\} \qquad \text{[definition of } FD_2 \text{ and step law of parallelism]}$$
$$= pass \to STOP \qquad \text{[step law of hiding]}$$

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

The above concepts and results are valid for all notions of refinement, although traces refinement is used in examples. In the next section, we consider the generation of tests specifically for traces refinement using fault domains.

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.

As explained in Section 2.1, the execution of a 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 $pass \in verd_{SUT}(T_T(t,a))$, 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 $verd_S\,UT(T_T(t,a)) = \{inc\}$, 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. What we need to is to refine the fault domain, and, in this way, reduce the set of processes that can potentially model the behaviour of the SUT. A trace implemented by the SUT must be a trace of the fault domain, as otherwise the SUT would not refine the fault domain, which is in contradiction with the definition of fault domain. Therefore, given the definition of traces refinement, we cannot use a trace implemented by the SUT to reduce, that is, refine the fault domain.

On the other hand, for a fault domain FD and trace t, with $t \notin traces[\![SUT]\!]$, we define a new (refined) 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. The set Σ contains all possible events, and so $RUN(\Sigma)$ accepts all possible events; it is the least refined process from the point of view of traces refinement. If we get to the end of t, then $NOTTRACE(t)$ prevents its last event from occurring. It, however, accepts any other event, and, after that, also behaves like $RUN(\Sigma)$.

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

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 such 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 $FDU(t)$ that excludes from a fault domain FD the trace t, we compose FD in parallel with $NOTTRACE(t)$ synchronising on the set Σ of all events.

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

Since the parallelism requires synchronisation on all events, it controls the occurrence of events as defined by $NOTTRACE(t)$. So, the fault domain defined by $FDU(t)$ excludes processes that perform t. We next establish that $FD \sqsubseteq_T FDU(t)$.

Theorem 5 $FD \sqsubseteq_T FDU(t)$

Proof

$$
\begin{aligned}
& traces[\![FDU(t)]\!] \\
&= traces[\![FD]\!] \cap traces[\![NOTTRACE(t)]\!] \\
&\qquad\qquad [\text{definition of } FDU(t) \text{ and of } traces[\![\_]\!] \text{ for parallelism}] \\
&\subseteq traces[\![FD]\!] \qquad [\text{property of set intersection } (A \cap B \subseteq A)]
\end{aligned}
$$

$\Box$

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 $S \sqsubseteq_T FDU(t) \sqsubseteq_T SUT$; thus, we can stop testing, since $S \sqsubseteq_T SUT$.

1: **procedure** TestGen(S, FD_{init}, SUT)
2: $FD := FD_{init}$
3: $failed :=$ **false**
4: $TS := \emptyset$
5: **while** $\neg\,(S \sqsubseteq_T FD) \wedge \neg\, failed$ **do**
6: Select a shortest $t \in (traces\,[\![FD]\!] \cap traces\,[\![S]\!]) \setminus TS$
7: **if** $initials(FD/t) \setminus initials(S/t) \neq \emptyset$ **then**
8: Select $a \in initials(FD/t) \setminus initials(S/t)$
9: $verd := ApplyTest(SUT, T_T(t, a))$
10: **if** $fail \in verd$ **then**
11: $failed :=$ **true**
12: **else if** $pass \in verd$ **then**
13: $FD := FDU(FD, t \frown \langle a \rangle)$
14: **else** ▷ that is, $verd = \{inc\}$
15: $FD := FDU(FD, t)$
16: **end if**
17: **else** ▷ that is, $initials(FD/t) \setminus initials(S/t) = \emptyset$
18: $TS := TS \cup \{t\}$
19: **end if**
20: **end while**
21: **return** $\neg\, failed$
22: **end procedure**

Fig. 1 Procedure for test generation

Based on these ideas, we now introduce a procedure TestGen for test generation. Its parameters are 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 of both the specification and the fault domain (line 6 of Figure 1). A test for 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. This is established below.

Theorem 6 *For $t \in traces\,[\![S]\!] \setminus traces\,[\![FD]\!]$, and any event a, the test $T_T(t, a)$ belongs to $Useless_{\sqsubseteq_T}(S, FD)$.*

Proof

$$t \in traces\,[\![S]\!] \setminus traces\,[\![FD]\!]$$

$$\Rightarrow t \notin traces\,[\![FD]\!] \qquad \text{[definition of set difference } (\setminus)]$$

$$\Rightarrow t \notin traces\,[\![SUT]\!] \qquad [FD \sqsubseteq_T SUT, \text{ so } traces\,[\![SUT]\!] \subseteq traces\,[\![FD]\!]]$$

$$\Rightarrow \forall s : traces\,[\![Execution^S_{SUT}(T_T(t,a))]\!] \mid s \neq \langle\rangle \bullet last(s) = inc$$
[definitions of $T_T(t,a)$, $Execution^S_{SUT}(T_T(t,a))$, and $traces\,[\![\_]\!]$]

$$\Rightarrow passes_T(S, SUT, T_T(t,a))$$ [definition of $passes_T(S, SUT, T)$]

$$\Rightarrow T_T(t,a) \in Useless_{\sqsubseteq_T}(S, FD)$$
[$T_T(t,a) \in Exhaust_T(S)$ and definition of $Useless_{\sqsubseteq_T}(S, FD)$]

□

Next in the procedure (line 7 of Figure 1), we check whether t has a continuation that is allowed by the fault domain FD, but is forbidden by S. For a process P with a trace s, the set $initials(P/s)$ includes the events on which P is ready to engage, after performing the events in s. If t has a forbidden continuation allowed by FD, we choose one of them and record it in a (line 8 of Figure 1). If not, t is not (or no longer) useful to construct tests, and is added to TS. This is because a forbidden continuation a that is also forbidden by FD is guaranteed to be forbidden by the SUT. Thus, testing for a is useless. This is established below.

Theorem 7 *For $t \in traces\,[\![S]\!] \cap traces\,[\![FD]\!]$, and a such that $a \in \Sigma \setminus initials(S/t)$ and $a \notin initials(FD/t) \setminus initials(S/t)$, the test $T_T(t,a)$ belongs to $Useless_{\sqsubseteq_T}(S, FD)$.*

Proof

$$t \in traces\,[\![S]\!] \cap traces\,[\![FD]\!] \wedge$$
$$a \in \Sigma \setminus initials(S/t) \wedge a \notin initials(FD/t) \setminus initials(S/t)$$

$$= t \in traces\,[\![S]\!] \cap traces\,[\![FD]\!] \wedge a \in \Sigma \setminus (initials(FD/t) \cup initials(S/t))$$
[$(A \setminus B) \setminus (C \setminus B) = A \setminus (B \cup (C \setminus B)) = A \setminus (B \cup C)$]

$$\Rightarrow t \in traces\,[\![FD]\!] \wedge a \notin initials(FD/t)$$ [$B \subseteq A \cap B$ and $B \cap (A \setminus B) = \emptyset$]

$$\Rightarrow t \frown \langle a\rangle \notin traces\,[\![FD]\!]$$ [definitions of $traces\,[\![P]\!]$ and $initials(P/t)$]

$$\Rightarrow t \frown \langle a\rangle \notin traces\,[\![SUT]\!]$$ [$FD \sqsubseteq_T SUT$]

$$\Rightarrow \forall s : traces\,[\![Execution^S_{SUT}(T_T(t,a))]\!] \mid s \neq \langle\rangle \bullet last(s) \in \{inc, passs\}$$
[definition of $T_T(t,a)$ and $Execution^S_{SUT}(T_T(t,a))$]

$$\Rightarrow passes_T(S, SUT, T_T(t,a))$$ [definition of $passes_T(S, SUT, T)$]

$$\Rightarrow T_T(t,a) \in Useless_{\sqsubseteq_T}(S, FD)$$
[$T_T(t,a) \in Exhaust_T(S)$ and definition of $Useless_{\sqsubseteq_T}(S, FD)$]

□

In lines 9 to 19, 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 and forbidden continuations a of *Counter* are sketched below in order of increasing length.

$$T_T(\langle\,\rangle, sub) = pass \to sub \to fail \to STOP$$

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

$$T_T(\langle add, add\rangle, add) = inc \to add \to inc \to add \to pass \to add \to fail \to STOP$$

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

$$T_T(\langle add, add, sub, sub\rangle, sub) = \ldots$$

This is, of course, an infinite set, arising from an infinite set of traces. We recall, 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, of course, depend on the particular SUT; we consider below one example: $SUT = add \to add \to STOP$. We note that, at no point, we use this knowledge of the SUT to select tests. That knowledge is used just to identify the result of the tests used in our procedure for illustration purposes.

In considering $\textsc{TestGen}(Counter, SUT, RUN(\Sigma))$, 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 $FD_1 = NOTTRACE(\langle sub\rangle) = add \to RUN(\Sigma)$. 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.

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

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

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

5 Generating test sets: termination

We assume that the set Σ of all events is finite. This is usual and essential for model checking, for instance. In this setting, a finite specification, that is, a specification with a finite set of traces is a straightforward case, since traces are themselves finite. It suffices to test with each trace and each forbidden continuation; if Σ is finite, there are finitely many forbidden continuations as well.

Our procedure, however, can still be useful, because useless tests may be applied if the fault domain is not considered. Our procedure can reduce the number of tests, which may still be very large in the general case.

In Section 5.1 below, we present a fully formal proof of termination for finite specifications. In doing so, we bring insight into the design of our procedure and pinpoint precisely the points where finiteness is required to establish termination. That proof also establishes that, if the procedure does terminate, it does so with the right result. This paves the way for us to discuss termination for a finite SUT in Section 5.2. Finally, in Section 5.3, we explain what needs to be done to achieve termination for an SUT with an infinite number of traces.

5.1 Finite specification

In our formal proof, we use a refinement calculus [24] and, in particular, adopt the mathematical notation of Z [37] of the calculus in [10]. This is convenient because that is the notation used in the formalisation of the CSP testing theory.

What we prove is that the procedure satisfies the following specification.

$$return : [FD_{init} \sqsubseteq_T SUT, return' = S \sqsubseteq_T SUT]$$

It specifies the value of a variable *return* that represents the result, true or false, returned by the procedure, using a pre and a postcondition. The precondition $FD_{init} \sqsubseteq_T SUT$ simply records that FD_{init} is indeed a fault domain. The postcondition $return' = S \sqsubseteq_T SUT$ establishes that the final value $return'$ of the variable *return* indicates whether the SUT is a traces-refinement of the specification S or not. Final values of variables are denoted using the primed names of the variables. The variables that can be changed are indicated in the frame: the list of variables that precedes the pre and postcondition in the specification. In this case, it is just *return*; the other variables of the procedure are local.

The proof here shows that the procedure terminates and establishes the postcondition, changing only *return*. A proof in a refinement calculus proceeds by applying algebraic refinement laws, which transform the specification to introduce the control constructs and assignments of the procedure. Here, the notion of refinement is not any of those of CSP, but that for sequential code. In the laws, we use the standard symbol $\sqsubseteq$. In this context, $P_1 \sqsubseteq P_2$ holds when P_2 has the same or a weaker precondition than P_1 and the same or a stronger postcondition. So, P_2 terminates whenever P_1 does and establishes its postcondition. A law application typically generates a verification condition that must be proved as part of proving refinement. The laws used here [10] are reproduced in Appendix A for convenience.

The first two refinement laws we apply are the Laws *vrbI* (variable introduction) and *seqcI* (sequential composition introduction), which declare the local variables, put them in the frame, so that they can be initialised and used, and break the specification into a sequence of two: the first is for the variable initialisation (lines 2 to 4), and the second is for the loop and the final assignment (lines 5 to 21). For that, we need to choose the postcondition of the first specification, which becomes the precondition of the second. Here, we need to use the loop invariant *inv* defined below. In the postcondition of the first specification, we use inv', to enforce it on the value of the updated variables. As expected, the initialisation needs to establish the invariant, and the loop can assume that it holds. The result is shown below.

$$
\begin{array}{l}
|[\ \mathbf{var}\ FD, \mathit{failed}, TS, \mathit{tSoFar} \bullet \\
\quad \mathit{return}, FD, \mathit{failed}, TS, \mathit{tSoFar} : [FD_{init} \sqsubseteq_T SUT, inv'];\\
\quad \mathit{return}, FD, \mathit{failed}, TS, \mathit{tSoFar} : [inv, \mathit{return}' = S \sqsubseteq_T SUT]\\
]|
\end{array}
$$

We introduce an extra local variable *tSoFar* to record the tests that have already been covered in the procedure to facilitate specification and proof. It can be removed from the code because it is never assigned to any other variable.

The definition of the invariant is as follows.

$$
\begin{array}{ll}
inv \mathrel{\widehat=} & \\
FD \sqsubseteq_T SUT \land \mathit{failed} = (\exists\, T : \mathit{tSoFar} \bullet \mathit{fail} \in \mathit{Apply}(SUT, T)) & (1)\\
\land & \\
\mathit{tSoFar} \subseteq \mathit{Exhaust}_T(S) & (2)\\
\land & \\
(\forall\, T : \mathit{tSoFar} \bullet \mathit{traces}\,[\![T]\!] \notin \mathit{traces}\,[\![FD]\!] \lor \mathit{fail} \in \mathit{Apply}(SUT, T) \lor \mathit{traces}\,[\![T]\!] \in TS) & \\
\land & (3)\\
(\forall\, t : TS \bullet t \in (\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \land \mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) = \emptyset) & (4)\\
\land & \\
\neg\,(S \sqsubseteq_T FD) \Rightarrow (\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS \neq \emptyset & (5)
\end{array}
$$

The invariant establishes that FD is a proper fault domain, that *failed* records whether a test T considered so far has failed (1), and essential properties of these tests. They are all in the exhaustive test set (2), and, moreover, for each such test T, there are three possibilities: the trace $\mathit{traces}\,[\![T]\!]$ used in T has already been eliminated from the fault domain, it has failed, or it is in TS (3). The role of TS is explained next (4). It contains the traces of FD and S that do not have continuations that can be used to construct useful tests (see Theorem 7). Finally, with (5) the invariant guarantees that, if the specification is not refined by the fault domain, then there are still tests that can be used (see Theorem 6).

With the Law *assigI* (assignment introduction), which refines a specification to an assignment, we can justify the initialisation of the local variables as shown in Figure 1 (lines 2 to 4). In addition, we initialise *tSoFar* with the set below.

$$
\mathit{tSoFar} := \{t : \mathit{traces}\,[\![S]\!] \setminus \mathit{traces}\,[\![FD_{init}]\!];\ a \mid a \notin \mathit{initials}(S/t) \bullet T_T(t, a)\}
$$

This assignment to *tSoFar* along with the other assignments satisfy the invariant because $FD_{init} \sqsubseteq SUT$ means that, for every $t : \mathit{traces}\,[\![S]\!] \setminus \mathit{traces}\,[\![FD_{init}]\!]$ and $a \notin \mathit{initials}(S/t)$, the corresponding test is in the exhaustive test set, and so we have (2), but gives an inconclusive verdict (Theorem 6). This establishes also (1) because *failed* is assigned *false*. In the case of (3), it follows from the fact that the trace of every such test is not in $\mathit{traces}\,[\![FD_{init}]\!]$. We finally note that, if $TS = \emptyset$, then (4) is trivial, and we know that $(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS$ is not empty because the empty trace is a trace of every process. So, (5) also holds.

Next, we apply Law *fassigI* (following assignment) to introduce the assignment $\mathit{return} := \neg\ \mathit{failed}$ at the end, to represent the returned value. This law splits a specification into another specification with the same frame and precondition, followed by an assignment. The new specification takes into account the assignment in its postcondition. With that, we are left with the specification below for the loop, which enforces in the postcondition that the result is $\neg\, \mathit{failed}$.

$$\mathit{return}, \mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\mathit{inv}, \neg\, \mathit{failed}' = S \sqsubseteq_T \mathit{SUT}]$$

To refine this to a loop, we need to rewrite the postcondition in terms of the invariant and the negation of the loop condition. For that, we use Law *sP* (strengthen postcondition). We also apply Law *cFR* (contract frame) to remove *return* from the frame, because this variable is not changed by the loop. The result is as follows.

$$\mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\mathit{inv}, \mathit{inv}' \wedge ((S \sqsubseteq_T \mathit{FD}') \vee \mathit{failed}')]$$

The application of *sP* requires us to prove that the new postcondition is stronger. To show that this is the case, we consider each of the cases identified in the negation of the loop condition. If $S \sqsubseteq_T \mathit{FD}'$, then $\mathit{FD}' \sqsubseteq_T \mathit{SUT}$ from the inv' (1) gives us $S \sqsubseteq \mathit{SUT}$, because refinement is transitive. From that, the CSP testing theory establishes that the SUT does not fail any of the tests of the exhaustive test set. Since (1) establishes that failed' captures whether any tests in that set have failed so far, failed' is false, and so $\neg\, \mathit{failed}' = S \sqsubseteq_T \mathit{SUT}$. In the case failed', on the other hand, (1) establishes that there is a test in the exhaustive test set that has failed. This ensures that $\neg\, (S \sqsubseteq_T \mathit{SUT})$. So, again, $\neg\, \mathit{failed}' = S \sqsubseteq_T \mathit{SUT}$.

To apply Law *itI* (iteration introduction) to introduce the loop, we need to define a variant *vrt* for the loop that guarantees that it terminates; we present it below. It is here, and only here, that we use the fact that S and Σ are finite.

$$\mathit{vrt} \mathrel{\widehat{=}} (\#\mathit{Exhaust}_T(S) + \#\mathit{traces}\,[\![S]\!]) - (\#\mathit{tSoFar} + \#\mathit{TS})$$

The variant is an expression whose value is decreased every step of a loop, but never below 0. Here, the number of traces and tests to be considered is bound by the sizes of the exhaustive test set and of the set of traces of S (some of which may have no tests). Finiteness of S and Σ ensures that these sets are finite, and so, their sizes are well defined. As the traces and tests are considered, they are added to *tSoFar* and *TS*, and so, the difference after the sizes of these sets are added, is decreased at every step. This difference never goes below 0, because the invariant guarantees that $\mathit{tSoFar} \subseteq \mathit{Exhaust}_T(S)$ with (2) and $\mathit{TS} \subseteq \mathit{traces}\,[\![S]\!]$ with (4).

Accordingly, after application of Law *itI*, we apply also Law *sP* to simplify the postcondition of the specification of the loop body and require just that the variant is decreased. We obtain the specification below.

while $\neg\, (S \sqsubseteq_T \mathit{FD}) \wedge \neg\, \mathit{failed}$ **do**
  $\mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\neg\, (S \sqsubseteq_T \mathit{FD}) \wedge \neg\, \mathit{failed} \wedge \mathit{inv}, \mathit{inv}' \wedge \mathit{vrt}' < \mathit{vrt}]$
end while

The precondition of the specification records the loop condition, which is true at the start of each step of the loop. We next use Laws *vrbI* and *seqCI* again to introduce and initialise a local variable t to hold the next trace under consideration (line 6). Law *assigI* is also used to introduce the assignment to t, which is removed from the frame with Law *cfR*, because this variable is no longer updated.

$$
\begin{array}{l}
|[\ \mathbf{var}\ \ t \bullet \\
\quad t := \mathit{shortest}(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS \\
\quad FD, \mathit{failed}, TS, \mathit{tSoFar} : [\\
\qquad \neg\,(S \sqsubseteq_T FD) \wedge \neg\,\mathit{failed} \wedge \mathit{inv} \wedge t \in \mathit{shortest}(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS, \\
\qquad \mathit{inv}' \wedge \mathit{vrt}' < \mathit{vrt}] \\
]|
\end{array}
$$

The invariant conjunct (5) guarantees that the set $(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS$ is not empty, and so the assignment to t is well defined.

The remaining specification is refined to the outer conditional (lines 7 to 19) using Law *altI* (alternation introduction). (We note that this law uses Dijkstra's notation of guarded commands [12] as usual in refinement calculi, but it is simple to choose guards to justify a standard conditional as we need.)

$$
\begin{array}{l}
\mathbf{if}\ \ \mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) \neq \emptyset\ \mathbf{then} \\
\quad FD, \mathit{failed}, TS, \mathit{tSoFar} : [\\
\qquad \mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) \neq \emptyset \wedge \\
\qquad \neg\,(S \sqsubseteq_T FD) \wedge \neg\,\mathit{failed} \wedge \mathit{inv} \wedge t \in \mathit{shortest}(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS, \\
\qquad \mathit{inv}' \wedge \mathit{vrt}' < \mathit{vrt}] \\
\mathbf{else} \\
\quad FD, \mathit{failed}, TS, \mathit{tSoFar} : [\\
\qquad \mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) = \emptyset \wedge \\
\qquad \neg\,(S \sqsubseteq_T FD) \wedge \neg\,\mathit{failed} \wedge \mathit{inv} \wedge t \in \mathit{shortest}(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS, \\
\qquad \mathit{inv}' \wedge \mathit{vrt}' < \mathit{vrt}] \\
\mathbf{end\,if}
\end{array}
$$

In each branch of the conditional, we have the original specification, with the precondition enriched to capture the case covered by that branch.

We first discuss the refinement of the else-branch, which can be tackled with the Law *assigI* to introduce the multiple assignment below.

$$\mathit{tSoFar}, TS := \mathit{tSoFar} \cup \{b \mid b \notin \mathit{initials}(S/t) \bullet T_T(t, b)\}, TS \cup \{t\}$$

A multiple assignment updates the variables in parallel. We recall that *tSoFar* is considered just for reasoning and can be eliminated at the end to obtain the procedure in Figure 1, where just *TS* is updated (line 18).

With this assignment, we need to prove that the invariant is maintained and the variant is reduced. For the invariant, we consider each conjunct in turn. In the case of (1), first of all, FD has not been changed. Second, the precondition states that $\neg\,\mathit{failed}$. So, we know that there are no failed tests in *tSoFar* before the assignment. Moreover, the new tests are useless, because b is not in the set $\mathit{initials}(FD/t) \setminus \mathit{initials}(S/t)$, which is empty according to the precondition (see Theorem 7). So, they do not fail either. For (2), the result is simple because t is a trace of S, according to the precondition, and b is a forbidden continuation.

For (3), we have $\mathit{traces}\,[\![T_T(t, b)]\!] = t$, of course, and $t \in TS \cup \{t\}$. So the new tests satisfy one of the disjuncts. For (4), the concern is just the new trace t added to *TS*, but the precondition guarantees that it satisfies the required properties.

Finally, (5) is an important result, because it guarantees that if we have not refined FD enough to get to a conclusion, then there must be more traces to generate tests. This is not an obvious result, and we prove it more formally, by contraction. We assume that $\neg\ S \sqsubseteq_T FD$ and suppose, by way of contradiction that $(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus (TS \cup \{t\}) = \emptyset$. We can then proceed as follows.

$$
\begin{array}{ll}
= (\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \subseteq (TS \cup \{t\}) & \text{[property of sets]} \\
\Rightarrow \forall\, u : \mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!] \bullet \mathit{initials}(FD/u) \subset \mathit{initials}(S/u) \lor u = t & \\
& \text{[property of } TS \text{ from } \mathit{inv}] \\
\Rightarrow \forall\, u : \mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!] \bullet \mathit{initials}(FD/u) \subset \mathit{initials}(S/u) & \\
& [\mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) = \text{from precondition}]
\end{array}
$$

This, however, allows us to prove $S \sqsubseteq_T FD$, that is, $\forall\, v : \mathit{traces}\,[\![FD]\!] \bullet v \in \mathit{traces}\,[\![S]\!]$, and we have a contradiction. We carry out this proof by induction on the trace v. For the empty trace, this is trivial, because it is a trace of every process. For the inductive case $v \frown \langle a \rangle$, we proceed as shown below.

$$
\begin{array}{ll}
v \frown \langle a \rangle \in \mathit{traces}\,[\![FD]\!] & \\
\Rightarrow v \in \mathit{traces}\,[\![FD]\!] & \text{[prefix closure of } \mathit{traces}\,[\![P]\!], \text{ for every } P] \\
\Rightarrow v \in \mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!] & \text{[induction hypothesis]} \\
\Rightarrow \mathit{initials}(FD/v) \subseteq \mathit{initials}(S/v) & \text{[result above]} \\
\Rightarrow v \frown \langle a \rangle \in \mathit{traces}\,[\![S]\!] & [v \frown \langle a \rangle \in \mathit{traces}\,[\![FD]\!]]
\end{array}
$$

As for the variant, we are increasing the size of TS, since the precondition guarantees that $t \notin TS$. So, the variant is decreased.

Going back to the specification of then-branch of the conditional above, we follow a similar sequence of law applications to introduce the local variables a and verd (Law vrbI) and their initialisations (Laws seqC and assigI) in lines 8 and 9, to be left with the specification for the inner conditional (lines 10 to 16). We use a predicate init below to capture the initialisation and abbreviate the specifications.

$$
\begin{array}{rl}
\mathit{init} \mathrel{\widehat{=}} & a \in \mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) \land \mathit{verd} = \mathit{ApplyTest}(SUT, T_T(t, a)) \land \\
& \mathit{tSoFar} = \mathit{TSF} \cup \{T_T(t, a)\} \land t \in \mathit{shortest}(\mathit{traces}\,[\![FD]\!] \cap \mathit{traces}\,[\![S]\!]) \setminus TS
\end{array}
$$

We note that init records also an update of tSoFar and the property of t. So, at this stage, we do also introduce the assignment $\mathit{tSoFar} := \mathit{tSoFar} \cup \{T_T(t, a)\}$ for convenience. In init, TSF stands for the value of tSoFar before this assignment. After contracting the frame of the inner conditional specification to record that we do not further update a and verd, we obtain the following result.

$$
\begin{array}{l}
FD, \mathit{failed}, TS, \mathit{tSoFar} : [\, \mathit{init} \land \neg\, (S \sqsubseteq_T FD) \land \neg\, \mathit{failed} \land \mathit{inv}[\mathit{TSF}/\mathit{tSoFar}], \\
\qquad\qquad\qquad\qquad\quad \mathit{inv}' \land \mathit{vrt}' < \mathit{vrt}[\mathit{TSF}/\mathit{tSoFar}]\,]
\end{array}
$$

The invariant in the precondition and the initial variant now refer to TSF.

To introduce the conditional with three branches, we again use Law *altI*.

$$
\begin{array}{l}
\textbf{if } \mathit{fail} \in \mathit{verd} \textbf{ then} \\
\quad \mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\, \mathit{fail} \in \mathit{verd} \land \\
\qquad\qquad \mathit{init} \land \neg (S \sqsubseteq_T \mathit{FD}) \land \neg \mathit{failed} \land \mathit{inv}[\mathit{TSF}/\mathit{tSoFar}], \\
\qquad\qquad \mathit{inv}' \land \mathit{vrt}' < \mathit{vrt}[\mathit{TSF}/\mathit{tSoFar}] \,] \\
\textbf{elseif } \mathit{pass} \in \mathit{verd} \textbf{ then} \\
\quad \mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\\
\qquad \mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\, \mathit{fail} \notin \mathit{verd} \land \mathit{pass} \in \mathit{verd} \land \\
\qquad\qquad \mathit{init} \land \neg (S \sqsubseteq_T \mathit{FD}) \land \neg \mathit{failed} \land \mathit{inv}[\mathit{TSF}/\mathit{tSoFar}], \\
\qquad\qquad \mathit{inv}' \land \mathit{vrt}' < \mathit{vrt}[\mathit{TSF}/\mathit{tSoFar}] \,] \\
\textbf{elseif } \mathit{pass} \in \mathit{verd} \textbf{ then} \\
\quad \mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\\
\qquad \mathit{FD}, \mathit{failed}, \mathit{TS}, \mathit{tSoFar} : [\, \mathit{fail} \notin \mathit{verd} \land \mathit{pass} \notin \mathit{verd} \land \\
\qquad\qquad \mathit{init} \land \neg (S \sqsubseteq_T \mathit{FD}) \land \neg \mathit{failed} \land \mathit{inv}[\mathit{TSF}/\mathit{tSoFar}], \\
\qquad\qquad \mathit{inv}' \land \mathit{vrt}' < \mathit{vrt}[\mathit{TSF}/\mathit{tSoFar}] \,] \\
\textbf{end if}
\end{array}
$$

We now apply Law *assigI* to each branch to obtain the right assignment. In each case, we need to prove that the assignment satisfies the postcondition, when the precondition holds and the other variables are not changed. Basically, we have to prove that the invariant is maintained and the variant is decreased.

For the first assignment, $\mathit{failed} := \mathit{true}$ (line 11), regarding the invariant we need to prove $\mathit{true} = \exists\, T : \mathit{TSF} \cup \{T_T(t,a)\} \bullet \mathit{fail} \in \mathit{Apply}(\mathit{SUT}, T)$ because of (1). This follows directly from *init* and $\mathit{fail} \in \mathit{verd}$. Also, because the precondition establishes that the invariant holds for TSF, rather than tSoFar, we need to prove (2) and (3) for $\mathit{tSoFar} = \mathit{TSF} \cup \{T_T(t,a)\}$. We have (2) because, according to *init*, we can conclude that t is a trace of S, and a is one of its forbidden continuations. For (3), again *init* and $\mathit{fail} \in \mathit{verd}$ gives us $\mathit{fail} \in \mathit{Apply}(\mathit{SUT}, T_T(t,a))$. Since (4) and (5) do not involve *failed* or *tSoFar*, they follow from the invariant in the precondition.

For the variant, we show that $\#\mathit{tSoFar} > \#\mathit{TSF}$, by showing $T_T(t,a) \notin \mathit{TSF}$. From the precondition, we know that $\neg\, \mathit{failed}$, so (1) in $\mathit{inv}[\mathit{TSF}/\mathit{tSoFar}]$ gives that all tests in TSF do not fail. Since *init* and the precondition give us that $T_T(t,a)$ has failed, then we get the required conclusion.

For the assignment $\mathit{FD} := \mathit{FDU}(\mathit{FD}, t \frown \langle a \rangle)$ (line 13), regarding the invariant, we need to establish $\mathit{FDU}(\mathit{FD}, t \frown \langle a \rangle) \sqsubseteq_T \mathit{SUT}$ because of (1). Since $\mathit{pass} \in \mathit{verd}$ ensures that $t \frown \langle a \rangle$ is not a trace of the SUT, this follows from Theorem 5 as explained in the previous section. For (1), we also observe that the precondition gives $\mathit{failed} = \mathit{false}$, and so we need $\neg\, \exists\, T : \mathit{tSoFar} \bullet \mathit{fail} \in \mathit{Apply}(\mathit{SUT}, T)$. This follows from $\mathit{failed} = \mathit{false} = \exists\, T : \mathit{TSF} \bullet \mathit{fail} \in \mathit{Apply}(\mathit{SUT}, T)$ in $\mathit{inv}[\mathit{TSF}/\mathit{tSoFar}]$, since *init* establishes $\mathit{tSoFar} = \mathit{TSF} \cup \{T_T(t,a)\}$ and the precondition guarantees that $T_T(t,a)$ does not fail. In the case of (2), the argument is as for the previous assignment. By Theorem 5, we know $\mathit{FDU}(\mathit{FD}, t \frown \langle a \rangle)$ has fewer traces than FD, so (3) for $\mathit{FDU}(\mathit{FD}, t \frown \langle a \rangle)$ is simple: it follows from (3) in *inv*.

Proof of (4) and (5) is not trivial, and provides insight as to why update of the fault domain does not affect the essential properties of TS. We tackle them more formally. For (4), we first establish the following, where we use $\leq$ for trace prefix.

$$
\begin{array}{ll}
t \frown \langle a \rangle \notin \mathit{traces}\,[\![S]\!] & [\text{from } \mathit{init}] \\
\Rightarrow \{ s \mid t \frown \langle a \rangle \leq s \} \cap \mathit{traces}\,[\![S]\!] = \emptyset & [\text{prefix closure of } \mathit{traces}\,[\![S]\!]]
\end{array}
$$

$$\Rightarrow \{ s \mid t \frown \langle a \rangle \leq s \} \cap TS = \emptyset \qquad \text{[property of } TS \text{ in } inv \text{ (4)]}$$
$$\Rightarrow \forall t : TS \bullet t \notin \{ s \mid t \frown \langle a \rangle \leq s \} \qquad \text{[property of sets]}$$
$$\Rightarrow \forall t : TS \bullet t \notin \{ s \mid t \frown \langle a \rangle \leq s \} \land t \in traces \llbracket FD \rrbracket \qquad \text{[property of } TS \text{ in } inv \text{ (4)]}$$
$$\Rightarrow \forall t : TS \bullet t \in traces \llbracket FDU(FD, t \frown \langle a \rangle) \rrbracket \qquad \text{[definition of } FDU(FD, t \frown \langle a \rangle)\text{]}$$

In addition, we can establish the property of $initials(FDU(FD, t \frown \langle a \rangle)/t)$ as follows.

$$\forall t : traces \llbracket FDU(FD, t \frown \langle a \rangle) \rrbracket \bullet initials(FDU(FD, t \frown \langle a \rangle)/t) \subseteq initials(FD/t)$$
$$\text{[since } FD \sqsubseteq_T FDU(FD, t \frown \langle a \rangle)\text{]}$$

$$\Rightarrow \forall t : traces \llbracket FDU(FD, t \frown \langle a \rangle) \rrbracket \bullet initials(FD/t) \setminus initials(S/t) = \emptyset \Rightarrow$$
$$initials(FDU(FD, t \frown \langle a \rangle)/t) \setminus initials(S/t) = \emptyset \qquad \text{[property of sets]}$$

So, we get the required result because (4) ensures $initials(FD/t) \setminus initials(S/t) = \emptyset$.

For (5), we note that since $t \frown \langle a \rangle \notin traces \llbracket S \rrbracket$, none of the extensions of these traces are in $traces \llbracket S \rrbracket$ either. These are the traces removed from the fault domain $FDU(FD, t \frown \langle a \rangle)$. So, $traces \llbracket FDU(FD, t \frown \langle a \rangle) \rrbracket \cap traces \llbracket S \rrbracket = traces \llbracket FD \rrbracket \cap traces \llbracket S \rrbracket$. The invariant then ensures that removing TS from this set does not make it empty.

For the variant, we again show that $\# tSoFar > \# TSF$ because $T_T(t, a) \notin TSF$. Here, we use (3) from $inv[TSF/tSoFar]$ and establish that $T_T(t, a)$ does not satisfy any of the identified properties of the elements of TSF. From $init$, we get $t \notin TS$ and $t \in traces \llbracket FD \rrbracket$, and $init$ and the precondition give us that $T_T(t, a)$ has passed.

For the last assignment (line 15), we introduce an assignment to $tSoFar$ as well.

$$tSoFar, FD := tSoFar \cup \{ b \mid b \notin initials(S/t) \bullet T_T(t, b) \}, FDU(FD, t)$$

Proof requires us to establish $FDU(FD, t) \sqsubseteq_T SUT$. Since neither $fail$ or $pass$ is in $verd$, then the verdict is inc, which ensures that t is not a trace of the SUT. So, the refinement follows from Theorem 5 as explained for the previous assignment. For (1), we also need $failed = false$ is equivalent to

$$\exists T : tSoFar \cup \{ b \mid b \notin initials(S/t) \bullet T_T(t, b) \} \bullet fail \in Apply(SUT, T)$$

This follows for $tSoFar$ as explained above for the previous assignment. For the new tests in $\{ b \mid b \notin initials(S/t) \bullet T_T(t, b) \}$, we note that t is not a trace of the SUT, so all of their verdicts is inc, just like for $T_T(t, a)$. In the case of (2), because the precondition ensures that $t \in traces \llbracket S \rrbracket$, we can conclude that the new tests are in the exhaustive test set. For (3), the argument is as for the previous assignment, but we also need to consider the new tests in $\{ b \mid b \notin initials(S/t) \bullet T_T(t, b) \}$ now added no $tSoFar$. Their traces are all t, which does not belong to $FDU(FD, t)$, so they all satisfy the first disjunct in the quantification. The proofs for (4) and (5) are very similar to those presented above for the previous assignment.

For the variant, the same argument used for the previous assignment applies. In this case, the test $T_T(t, a)$ does not pass: $init$ and the precondition mean that it is inconclusive, but what matters is that it does not fail.

To complete, we note that there is no assignment of an expression depending on $tSoFar$ to any other variable. So, it is what is technically called in refinement calculi an auxiliary variable, and can be eliminated as predicted.

5.2 Finite SUT

The proof in the previous section establishes that our procedure gives the correct result if it terminates, and that it terminates if the specification is finite. We now discuss a scenario where the specification is not finite, but the SUT is.

We note, first of all, that if the SUT is incorrect, that is, it does not trace refine the specification, the procedure always terminates. The fact that, in this case, it returns false is a correctness issue already addressed in the previous section. In the sequel, we denote by $pref(t)$ all prefixes of t, that is, $pref(t) = \{s : \Sigma^* \mid s \leq t\}$.

Theorem 1 *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*. □

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

Example 7 We consider the specification

$$UNBOUNDED = a \rightarrow UNBOUNDED \;\Box\; b \rightarrow STOP$$

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 $FD_1 = FDU(FD_{init}, \langle b \rangle)$ expanded 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. We then 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 $FD_2 = FDU(FD_1, \langle a, b\rangle)$.

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

If we proceed, we observe that the refined fault domains are always of the form

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

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, as proved in Theorem 1, the procedure terminates. $\Box$

In the next example, we illustrate a situation where the procedure does terminate.

Example 8 We now consider *Counter* from Example 1 and explain why our procedure terminates for its correct finite implementations.

First, we recall that our procedure uses traces of increasing length for deriving and applying tests (line 6), and note that 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 the process $Counter/t$.

Two possibilities are $Counter/t = Counter$ or $Counter/t = Counter2$, in which case 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 to remove t (line 15) from it and, thus, exclude f 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 explained above. 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.

Eventually, the set of traces extending t which remains in the fault domain is empty. More formally, in terms of the loop invariant, $(traces\,[\![FD]\!] \cap traces\,[\![S]\!]) \setminus = \emptyset$, and so according to (5), we get $S \sqsubseteq_T FD$, and the procedure terminates (line 5).

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. $\Box$

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

Given traces r and t, we say that 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 $ppref(t)$ the set of 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 $traces \llbracket S \rrbracket_k$ of the traces of S of length k. Formally, $traces \llbracket S \rrbracket_k = \{t : traces \llbracket S \rrbracket \mid \#t = k\}$. Another subset $hfc(S, FD)$ of traces of S includes those for which there is at least one forbidden continuation that takes into account the fault domain.

$$hfc(S, FD) = \{t : traces \llbracket S \rrbracket \mid initials(FD/t) \setminus initials(S/t) \neq \emptyset\}$$

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.

Theorem 2 *For a specification S and a fault domain FD_{init}, if, for any finite set of traces $P \subseteq traces \llbracket S \rrbracket$, there exists a $k \geq 0$, such that, for each $r \in traces \llbracket S \rrbracket_k$, 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 Theorem 1, the procedure terminates. We, therefore, assume that $S \sqsubseteq_T SUT$, and so $traces \llbracket SUT \rrbracket \subseteq traces \llbracket S \rrbracket$.

Let $k \geq 0$ be such that, for each $r \in traces \llbracket S \rrbracket_k$, we have

$$((pref(r) \setminus traces \llbracket SUT \rrbracket) \cap hfc(S, FD_{init})) \neq \emptyset$$

This k is larger than the size of the largest trace of SUT, since otherwise, if $r \in traces \llbracket SUT \rrbracket$, then $pref(r) \setminus traces \llbracket SUT \rrbracket$ is empty, and if $r \notin traces \llbracket SUT \rrbracket$, the set $hfc(S, FD_{init})$ is empty. Moreover, such a k exists because $traces \llbracket SUT \rrbracket$ is finite.

Let now $Q = (pref(traces \llbracket S \rrbracket_k) \setminus traces \llbracket SUT \rrbracket) \cap hfc(S, FD_{init})$ and and also let $M = minimals(Q)$. Let $p \in traces \llbracket SUT \rrbracket$. Let $r \in traces \llbracket S \rrbracket_k$ be such that $p \leq r$. There is at least one $s \in pref(r)$, such that $p \leq s$ and $s \in hfc(S, FD_{init})$ because $((pref(r) \setminus traces \llbracket SUT \rrbracket) \cap hfc(S, FD_{init})) \neq \emptyset$. 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 $traces \llbracket SUT \rrbracket \subseteq pref(M)$ since p is arbitrary.

For each $t \in M$, there exists at least $a \in initials(FD_{init}/t) \setminus initials(S/t)$, since $t \in hfc(S, FD_{init})$. Therefore, if the test $T_T(t, a)$ is applied to the SUT, the verdict is inc, since $t \notin traces \llbracket SUT \rrbracket$ because $t \in M \subseteq Q$ and $Q \cap traces \llbracket SUT \rrbracket = \emptyset$. In this case, the fault domain is updated to remove t from it.

If all tests derived for each $t \in M$ are applied, we obtain a fault domain FD such that $traces \llbracket FD \rrbracket \subseteq pref(M)$. 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. Since

$$pref(M) \subseteq Q \subseteq hfc(S, FD_{init}) \subseteq traces \llbracket S \rrbracket$$

it follows that $traces \llbracket FD \rrbracket \subseteq traces \llbracket S \rrbracket$, that is, $S \sqsubseteq_T SUT$. $TestGen(S, FD_{init}, SUT)$ then terminates, with $failed = false$, and the result is $true$. $\Box$

One scenario where the conditions in Theorem 2 hold is if there is an event in the alphabet that 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.

5.3 Infinite SUT

In the cases that we have considered so far, the number of traces of the specification or of the SUT is finite. These scenarios are relevant in practice, when systems have a set of paths from initiation to conclusion, like. Most systems, however, do have an infinite behavior and thus an infinite set of traces.

There are some approaches to deal with this situation, all of them rendering the set of processes in the fault domain finite. For instance, the complexity of the SUT processes can be somehow constrained, limiting the number of states they may have. This is what is done in classical fault-domain based testing from Finite State Machines, when the SUT is assumed to have a maximum number of states. Albeit interesting, as previously explained, the correlation between the states and the traces of a CSP process is not trivial. So, this approach cannot be easily integrated with the use of our procedure.

Another approach that is usually employed to render the test activity always finite is to set an upper bound on the length of traces considered during the test. Similar solutions have been used in [6] and [36] for limiting the tests by setting the maximum length of the trace in the test. We observe that this approach is usual, but is indeed a tradeoff between completeness and finiteness; while the test execution is now always finite (provided that the event alphabet is finite), the pass verdict is only a partial answer which should be read as correct up that trace length. Further conclusions depend on a regularity hypothesis [16].

In the procedure, the adjustment to be done is in the loop condition in line 5 of Figure 2. If k is the maximum length of the traces to be used for test generation. in 2.

In revisiting the proof in Section 5.1, it can proceed in much the same way, with the size of the set $(traces\ [\![FD]\!]_k \cap traces\ [\![S]\!]_k) \setminus TS$ as the variant. The specification, however, needs to be modified. We can, for example, add a precondition that states that all traces of the SUT that have more than k elements are in the specification. This is, in some sense, a way of restricting the fault domain.

6 Tool support and case studies

We have developed a prototype tool that implements our procedure. It and all examples presented here are available at `www.github.com/adenilso/CSP-FD-TGen`.

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. Checking whether the specification refines the fault domain (line 5). This is a straightforward refinement check in FDR.

1: **procedure** TESTGEN(S, FD_{init}, SUT)
2: $\quad FD := FD_{init}$
3: $\quad$ *failed* := **false**
4: $\quad TS := \emptyset$
5: $\quad$ **while** $\neg\, (S \sqsubseteq_T FD) \wedge \neg\, \mathit{failed} \wedge (\mathit{traces}\, [\![FD]\!]_k \cap \mathit{traces}\, [\![S]\!]_k) \setminus TS \neq \emptyset$ **do**
6: $\qquad$ Select a shortest $t \in (\mathit{traces}\, [\![FD]\!] \cap \mathit{traces}\, [\![S]\!]) \setminus TS$
7: $\qquad$ **if** $\mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) \neq \emptyset$ **then**
8: $\qquad\quad$ Select $a \in \mathit{initials}(FD/t) \setminus \mathit{initials}(S/t)$
9: $\qquad\quad \mathit{verd} := \mathit{ApplyTest}(SUT, T_T(t, a))$
10: $\qquad\quad$ **if** $\mathit{fail} \in \mathit{verd}$ **then**
11: $\qquad\qquad$ *failed* := **true**
12: $\qquad\quad$ **else if** $\mathit{pass} \in \mathit{verd}$ **then**
13: $\qquad\qquad FD := FDU(FD, t \frown \langle a \rangle)$
14: $\qquad\quad$ **else** $\qquad\qquad\qquad\qquad \triangleright$ that is, $\mathit{verd} = \{\mathit{inc}\}$
15: $\qquad\qquad FD := FDU(FD, t)$
16: $\qquad\quad$ **end if**
17: $\qquad$ **else** $\qquad\qquad\qquad\qquad \triangleright$ that is, $\mathit{initials}(FD/t) \setminus \mathit{initials}(S/t) = \emptyset$
18: $\qquad\quad TS := TS \cup \{t\}$
19: $\qquad$ **end if**
20: $\quad$ **end while**
21: $\quad$ **return** $\neg\, \mathit{failed}$
22: **end procedure**

Fig. 2 Modified procedure for test generation (line 5).

2. Computing forbidden continuations (lines 7 and 8). For instance, to compute the forbidden continuations of a process S after a trace t, that is, 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 the whole set of events Σ. It is defined as follows.

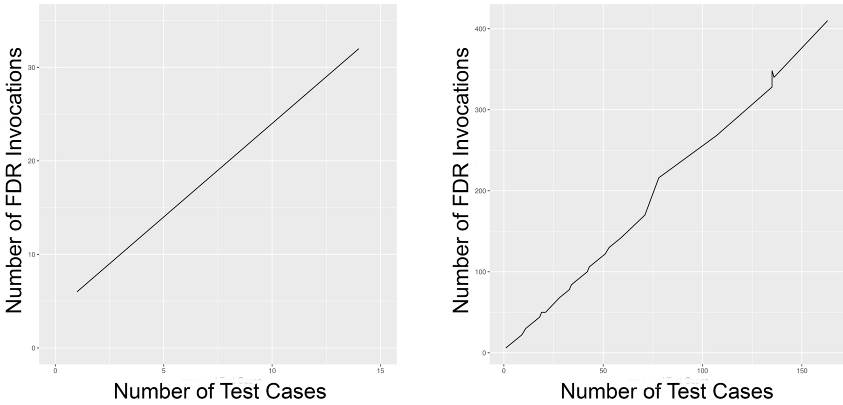
$$\mathit{TTHENANY}(\langle\rangle) = \Box\, e : \Sigma \bullet e \to \mathit{STOP}$$
$$\mathit{TTHENANY}(\langle a \rangle \frown t) = a \to \mathit{TTHENANY}(t)$$

   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 $\mathit{initials}(S/t)$.
3. Computing initials (lines 7 and 8). We use the same check above, and obtain $\mathit{initials}(S/t)$ by collecting the events in Σ for which no 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 carried out two significant case studies, the Transputer-based sensor for autonomous vehicles in [30], 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 [25] to assess the deadlock detection of a prototype model checker for *Circus* [8], a combination of CSP and Z [37].

For each case study, we have randomly generated 1000 finite SUTs. The experimental results confirm that all incorrect SUTs are identified and the procedure



ERS Autonomous Vehicles

Fig. 3 Number of Test Cases *versus* Number of Invocations to FDR

ERS Autonomous Vehicles

Fig. 4 Number of Test Cases *versus* Execution Time

terminates for all finite SUTs identified in Theorem 2. The CSP model for the sensor, the ERS, and other examples are in `www.github.com/adenilso/CSP-FD-TGen`. The data for the SUTs used in these case studies are also available.

We have measured the effort of generating and executing the tests for our case studies. Most of the time is consumed by the invocations of FDR for the tasks enumerated above. In Figure 3, we show the relation between the number of test cases executed and the number of FDR invocations for each case study. In Figure 4, we present the execution time variation related to the number of test cases. From both figures, we can conclude that the main component of the execution time is the invocation of the FDR tool. In the case studies, both the execution time and the number of invocation grows linearly with the number of test cases.

We have also run the tool with a set of small examples to assess the expected execution time of the tool. The examples were collected from [33], and illustrate many aspects of CSP operators. 1000 mutants were generated and tested. The

Example	Short Description	Average Time (Seconds)	Max Time (Seconds)	Max. Num. Test Cases
Counter	Running Example	0.89	47.91	9
Cross	Train Gate Control	0.87	33.39	19
Pump	Gas Dispenser Simulation	1.24	42.17	58
Purchase	Simple Transaction Simulation	0.96	50.02	33
Shopping	Security Control Simulation	0.82	70.13	27

Table 1 Time Execution and Test Cases for Case Studies.

verdict was correct in every case. We can observe that, even though some executions took almost one minute to complete, the average execution time is about one second or less (See 1). All examples and respective mutants are also available in the above mentioned repository.

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 [7]. The motivation is exactly work with testing. It is worth investigating how fault domains can be used to generate finite test sets for these notions of conformance.

A Refinement laws

Law *altI* Alternation introduction

$w : [pre, post]$

$\sqsubseteq$ *altI*

if $\square\, i \bullet g_i \,\&\, w : [g_i \wedge pre, post]$ **fi**

provided $pre \Rightarrow (\bigvee i \bullet g_i)$

Syntactic Restrictions

- Each g_i is a well-scoped predicate;
- No g_i has free dashed variables;
- $\{ i \bullet g_i \}$ is non-empty.

Law *assigI* Assignment introduction

$$w, vl : [pre, post]$$
$$\sqsubseteq \ assigI$$
$$vl := el$$

provided $pre \Rightarrow post[el/vl'][\_/']$

Syntactic Restrictions
- vl contains no duplicated variables;
- vl and el have the same length;
- el is well-scoped and well-typed;
- el has no free dashed variables;
- The corresponding variables of vl and expressions of el have the same type.

Law *cfR* Contract frame

$$w, x : [pre, post]$$
$$\sqsubseteq \ cfR$$
$$x : [pre, post[w/w']\ ]$$

Syntactic Restriction The variables of w are not in x.

Law *fassigI* Following assignment introduction

$$w, vl : [pre, post]$$
$$\sqsubseteq \ fassigI$$
$$w, vl : [pre, post[el[w', vl'/w, vl]/vl']\ ]\ ;\ \ vl := el$$

Syntactic Restrictions
- vl contains no duplicated variables;
- vl and el have the same length;
- el is well-scoped and well-typed;
- el has no free dashed variables;
- The corresponding variables of vl and expressions of el have the same type.

Law *itI* Iteration introduction

$$w : [inv, inv[w'/w] \wedge \neg\,(\bigvee i \bullet g_i[w'/w])]$$
$$\sqsubseteq \ itI$$
$$\mathbf{do} \ \square\ i \bullet g_i \ \&\ w : [inv \wedge g_i, inv[w'/w] \wedge 0 \leq vrt[w'/w] < vrt]\ \mathbf{od}$$

Syntactic Restrictions
- vrt is a well-scoped and well-typed integer;
- Each g_i and vrt have no free dashed variables. expression.

Law *vrbI* Variable introduction

$w : [pre, post]$

$= vrbI$

$[\![\, \mathbf{var}\ dvl \bullet vl, w : [pre, post]\ ]\!]$

where *dvl* declares the variables of *vl*.

Syntactic Restrictions
- *dvl* is well-scoped and well-typed;
- The variables of *vl* and vl' are not free in $w : [pre, post]$ and are not dashed.

Law *seqcI* Sequential composition introduction

$w, x : [pre, post]$

$\sqsubseteq seqcI$

$w : [pre, mid[w'/w]\ ]\ ;\ \ w, x : [mid, post]$

Syntactic Restrictions
- *mid* is well-scoped and well-typed;
- *mid* has no free dashed variables;
- No free variable of *post* is in w.

Law *seqcI* Sequential composition introduction

$w, x, y!, z! : [pre, post]$

$\sqsubseteq seqcI$

$[\![\, \mathbf{con}\ dcl \bullet w, y! : [pre, mid]\ ;\ \ w, x, y!, z! : [mid[cl/w][\_/'], post[cl/w]\ ]\ ]\!]$

where *dcl* declares the constants of *cl*.

Syntactic Restrictions
- *mid* is well-scoped and well-typed;
- The names of *cl* and cl' are not free in *mid* and $w, x, y!, z! : [pre, post]$;
- *cl* and w have the same length;
- The constants of *cl* have the same type as the corresponding variables of w.

Law *sP* Strengthen postcondition

$w : [pre, post]$

$\sqsubseteq sP$

$w : [pre, npost]$

provided $pre \wedge npost \Rightarrow post$

Syntactic Restriction *npost* is well-scoped and well-typed.

Acknowledgements

The authors would like to thank financial support from Royal Society (Grant NI150186), FAPESP (Grant 2013/07375-0) and EPSRC (Grant EP/M025756/1). The authors also are thankful to Marie-Claude Gaudel, for useful discussions in an early version of this paper. No new primary data was generated.

References

1. A. Alberto, A. L. C. Cavalcanti, M.-C. Gaudel, and A. Simao. Formal mutation testing for *Circus*. *Information and Software Technology*, 81:131–153, 2017.
2. Z. Andrews, R. Payne, A. Romanovsky, A. Didier, and A. Mota. Model-based development of fault tolerant systems of systems. In *Systems Conference (SysCon), 2013 IEEE International*, pages 356–363, April 2013.
3. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *9th International Conference on Formal Engineering Methods*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer-Verlag, 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 *Fundamental Approaches to Software Engineering*, volume 8441 of *Lecture Notes in Computer Science*, pages 415–429, 2014.
6. A. L. C. Cavalcanti and M.-C. Gaudel. Test selection for traces refinement. *Theoretical Computer Science*, 563(0):1 – 42, 2015.
7. A. L. C. Cavalcanti and R. M. Hierions. Testing with Inputs and Outputs in CSP. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 359–374, 2013.
8. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146–181, 2003.
9. A. L. C. Cavalcanti and A. Simão. Fault-based testing for refinement in CSP. In N. Yevtushenko, A. R. Cavalli, and H. Yenigün, editors, *29th IFIP WG 6.1 International Conference on Testing Software and Systems*, volume 10533 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2017.
10. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.
11. T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
13. KA El-Fakih, Rita Dorofeeva, NV Yevtushenko, and GV Bochmann. Fsm-based testing from user defined faults adapted to incremental and mutation testing. *Programming and Computer Software*, 38(4):201–209, 2012.
14. S. Fujiwara and G. von Bochmann. Testing non-deterministic state machines with fault coverage. In *IFIP TC6/WG6.1 4th Int. Wshop on Protocol Test Systems IV*, pages 267–280. North-Holland, 1991.
15. T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.
16. R M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Transactions on Software Engineering and Methodology*, 11(4):427–448, 2002.
17. Robert M. Hierons and Hasan Ural. Optimizing the length of checking sequences. *IEEE Trans. on Computers*, 55(5):618–629, 2006.
18. Wen-ling Huang and Jan Peleska. Exhaustive model-based equivalence class testing. In *Testing Software and Systems - 25th IFIP WG 6.1 International Conference, ICTSS 2013, Istanbul, Turkey, November 13-15, 2013, Proceedings*, pages 49–64, 2013.
19. I. Koufareva, Alexandre Petrenko, and Nina Yevtushenko. Test generation driven by user-defined fault models. In *Testing of Communicating Systems: Method and Applications, IFIP TC6 12th International Workshop on Testing Communicating Systems, September 1-3, 1999, Budapest, Hungary*, pages 215–236, 1999.
20. G. Luo, G. v. Bochmann, and A. Petrenko. Test selection based on communicating non-deterministic finite-state machines using a generalized Wp-method. *IEEE Transactions on Software Engineering*, 20(2):149–162, February 1994.
21. A. J. R. G. Milner. *A Calculus of Communicating Systems*, volume 92. Springer Verlag, 1980.
22. Alan Moraes, Wilkerson de L. Andrade, and Patrícia D. L. Machado. A family of test selection criteria for timed input-output symbolic transition system models. *Sci. Comput. Program.*, 126:52–72, 2016.

23. L. J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, Aug 1990.
24. C. C. Morgan. *Programming from Specifications.* Prentice-Hall, 2nd edition, 1994.
25. Alexandre Mota, Adalberto Farias, Andre Didier, and Jim Woodcock. Rapid prototyping of a semantically well founded Circus model checker. In *Software Engineering and Formal Methods*, volume 8702 of *LNCS*, pages 235–249. Springer, 2014.
26. S. Nogueira, A. C. A. Sampaio, and A. C. Mota. Test generation from state based use case models. *Formal Aspects of Computing*, 26(3):441–490, 2014.
27. J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. In *Formal Methods Europe, Industrial Benefits and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, 1996.
28. A. Petrenko and N. Yevtushenko. Testing from partial deterministic FSM specifications. *IEEE Trans. on Computers*, 54(9), 2005.
29. Alexandre Petrenko, Gregor von Bochmann, and Ming Yu Yao. On fault coverage of tests for finite state specifications. *Computer Networks and ISDN Systems*, 29(1):81–106, 1996.
30. 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.
31. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321–340, 2003.
32. A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2011.
33. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Wiley, 2000.
34. S. Schneider and H. Treharne. Communicating B Machines. In D. Bert, J. Bowen, M. Henson, and K. Robinson, editors, *ZB'2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 416–435, 2002.
35. Jan Tretmans. Test generation with inputs, outputs, and quiescence. In *TACAS'96*, volume 1055 of *LNCS*, pages 127–146. Springer, 1996.
36. Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. Fault-based conformance testing in practice. *Int. J. Software and Informatics*, 3(2-3):375–411, 2009.
37. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.
38. Yuen Tak Yu and Man Fai Lau. Fault-based test suite prioritization for specification-based testing. *Information and Software Technology*, 54(2):179–202, 2012.