Testing for Refinement in CSP

Ana Cavalcanti[1] and Marie-Claude Gaudel[2]

[1] University of York, Department of Computer Science
York YO10 5DD, UK
[2] LRI, Université de Paris-Sud and CNRS
Orsay 91405, France

Abstract. CSP is a well-established formalism for modelling and verification of concurrent reactive systems based on refinement. Consolidated denotational models and an effective tool have encouraged much work on algebraic reasoning and model checking. Testing techniques based on CSP, however, have not been widely explored, and in this paper we take a first step by instantiating Gaudel et al's theory of formal testing to CSP. We identify the testability hypothesis that we consider necessary to use CSP models as a basis for testing. We also define test sets that we prove to be exhaustive with respect to traces and failures refinement, and consider optimisations, inputs and outputs, and selection strategies. Our results are proved in terms of the CSP denotational models; they are a sound foundation for the development of test-generation techniques.

1 Introduction

It is well accepted that formal specifications can be useful bases for software testing; we refer to [6, 11, 2, 3, 1, 17] among many other pioneering papers and surveys. In spite of that, testing based on CSP [26, 16], which is a popular formal notation for specification and verification of concurrent systems, has not been widely explored. In this paper, we establish the foundations of CSP-based testing by instantiating a well-established theory of formal testing.

Even though it has been recognised for a while that formal models can bring much to software testing, embedding implementation testing within a formal framework is far from being obvious. In this case, we test a system: a system is not a formula, even if it can be (partially) described as such. Thus, testing is related to, but very different from proof of correctness based on the program text using, for example, an assertion technique. Similarly, testing is different from model checking, where verifications are based on a known model of the system: when testing, the model corresponding to the system under test is unknown. If it was known, testing would not be necessary... Moreover, it is sometimes difficult to observe the state of the system under test [6, 12, 17]. These points have been successfully circumvented in several testing methods that are based on formal specifications (or models) and on conformance relations that precisely state what it means for a system under test to satisfy a specification [7–9, 31, 14, 13, 20, 18].

The gap between systems and models is generally taken into account by explicit assumptions on the systems under test [6, 1, 17] that are called "testability hypotheses" in [1] or "test hypotheses" in [4]. Such assumptions are fundamental in the proof that the success of the test set derived from the specification establishes the conformance relation. Moreover, they provide hints on complementary tests or proofs that may be necessary to ensure this equivalence.

CSP provides, in addition to a formal semantics of communicating processes, formal definitions of notions of refinement similar to the conformance relations used in specification-based testing or model-based testing. Peleska and Siegel [24, 25] have already studied and applied CSP-based testing. They have not, however, addressed the issue of the gap between the system under test and the CSP model that it defines. The practical test sets they propose in [25] are inspired by, but not in direct correspondence with, their theoretical definitions.

Schneider [27] defines a partition that classifies refusable and non-refusable events, and high-level and low-level events, for the purposes of modelling security applications in CSP. In that work, two conformance relations based on testing are defined, and Schneider shows how model checking can be used to establish such relations. In our work, on the other hand, we are interested in refinement.

More recently, CSP has been used to formalise a notion of conformance traditionally associated with input-output labelled transition systems [22]. This work goes well beyond ours, in that it provides a technique and a tool for generation and selection of tests based on the use of FDR, the CSP refinement model checker. Our definition of a test case, however, is similar to theirs. We believe that the results on exhaustiveness of test sets and factorisation that we present here are relevant to further justify some of the definitions in [22].

The work in [29] recognises the potential impact of the assumptions about the interaction of a system with its environment on refinement; it aims at supporting the validation of (implicit) assumptions. For that, mutation testing techniques are applied to a CSP model of the system, and the mutants that satisfy the properties of interest are used as a basis for the clarification of requirements.

In this paper, we state the testability hypotheses that are associated with the use of CSP. Moreover, we formalise accurately the kind of observations (traces and refusals) that must be done when performing test experiments derived from a CSP specification. It leads to a novel formulation of the tests and of their verdicts. We give algebraic proofs that getting the right observations when running these tests is equivalent to establishing traces or failures refinement.

In the next section, we give a brief introduction to CSP, and in Section 3 we discuss the consequences of using a process algebra as a basis for testing. In Section 4, we introduce our testability hypotheses and their consequences, and in Section 5, we give our exhaustive test sets. In the next three sections, we discuss possible optimisations of our tests, test selection criteria, and the issues raised by inputs and outputs. Proofs of our results can be found in [5]. We conclude in Section 9 with a summary, and a discussion of related and future work.

2 A few things on CSP

Here we briefly recall the syntax of CSP, and some important points of its semantics and notions of refinement. More information can be found in [26].

2.1 Main CSP operators

In CSP, a system is modelled as a process that can interact with its environment via a number of events. CSP models describe a collection of processes and their patterns of interactions. The unit of interaction is an event which processes perform and on which they may synchronise; the *occurrence* of events is atomic. The set of (external) events of a process P is denoted αP.

There are two basic processes: $STOP$ is a deadlocked process, and $SKIP$ is the terminating process. The process $a \rightarrow P$ can perform an event a and then behave as P. The external choice, $P_1 \Box P_2$, is initially prepared to behave either as P_1 or as P_2, with the choice being made on occurrence of the first event. Nondeterminism is modelled by an internal choice, $P_1 \sqcap P_2$, which is a process that can arbitrarily choose to behave as either P_1 or P_2.

Processes can also be combined in sequence, using the operator $;$, or in parallel. CSP provides a number of operators for parallelism; here we use the alphabetised parallelism: $P_1 \llbracket A \rrbracket P_2$, which executes P_1 and P_2 concurrently, requiring that they synchronise on events that are in the set A.

Events can be external, that is, observable and controllable by the environment, or internal. Using the hiding operator, like in $P \setminus A$, we define a process that behaves like P, but whose events in the set A are internal.

A simple example, which we use later on, is given below: a process $Counter_2$ that counts from 0 to 2, and is defined in terms of the processes C_1 and C_2.

$$\begin{aligned}
&Counter_2 = add \rightarrow C_1\\
&C_1 = add \rightarrow C_2 \Box sub \rightarrow Counter_2\\
&C_2 = sub \rightarrow C_1
\end{aligned}$$

The events in $\alpha Counter_2 = \{add, sub\}$ model requests to add or subtract.

In CSP, inputs and outputs are not primitive concepts; they are modelled using events whose names are composite. A classical example is a copying process *Replicator*, which takes an input x from a channel c and sends it back.

$$Replicator = c?x \rightarrow c!x \rightarrow C$$

If we assume that the type of x is the rather small set $\{\,0, 1, 2\,\}$, then we have the events $c.0$, $c.1$, and $c.2$. In this case, the above definition of *Replicator* is just an abbreviation for the following definition in terms of these basic events.

$$Replicator = (c.0 \rightarrow c.0 \rightarrow C) \Box (c.1 \rightarrow c.1 \rightarrow C) \Box (c.2 \rightarrow c.2 \rightarrow C)$$

An input $c?x \rightarrow P(x)$ is an abbreviation for a possibly infinite external choice over processes $c.v \rightarrow P(v)$, for all possible values v for x. An output event $c!e$, on the other hand, is just another name for $c.v$, where v is the value of e.

2.2 Semantics and refinement

There are three well-established semantic models of CSP: the traces, the (stable) failures, and the failures-divergences models. The traces model characterises a process P by its set $traces(P)$ of traces: finite sequences of events which it can perform. It is a subset of $((\alpha P) \cup \{ \checkmark \})^*$; A^* is the set of finite sequences of elements of A. The special event $\checkmark$ records termination. The empty trace is $\langle\,\rangle$. The set of all events, except only for $\checkmark$, is Σ; the set including $\checkmark$ is denoted $\Sigma^{\checkmark}$. In the sequel, we sometimes consider traces as processes: the finite trace $a_1, a_2, \ldots$ corresponds to the process $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow SKIP$.

As usual we write P/s to describe the behaviour of the process P after one of its traces s, and $initials(P/s)$ for the set of events performable by P after s. In fact, *initials* is defined for any process P [26, page 197], and can be characterised in terms of its traces as $initials(P) = \{ a : \Sigma^{\checkmark} \mid \langle a \rangle \in traces(P)$.

For a trace s of a process P and a subset $A = \{a_1, \ldots, a_n\}$ of αP, the pair (s, A) is a failure for P if, and only if, after performing s, P may refuse all events of A: in other words, $P \llbracket \alpha P \rrbracket (s;\ (a_1 \rightarrow P_1 \Box \ldots \Box a_n \rightarrow P_n)$ may deadlock just after s. For $Counter_2$, the set of failures includes the following elements.

$$\begin{array}{l}(\langle\,\rangle, \{ sub \}), (\langle\,\rangle, \emptyset), (\langle add \rangle, \emptyset),\\ (\langle add, add \rangle, \{ add \}), (\langle add, add \rangle, \emptyset), (\langle add, sub \rangle, \{ sub \}), (\langle add, sub \rangle, \emptyset),\\ (\langle add, add, sub \rangle, \emptyset), \ldots\end{array}$$

The set $failures(P)$ containing all failures of P is subset closed: for instance, if P may deadlock when the choice among the events $\{a, b\}$ is proposed by its environment after a trace s, it may deadlock as well if only a or b is proposed. The traces of P can be defined from its failures: $traces(P) = \{ t : \Sigma^{\checkmark *} \mid (t, \emptyset) \in failures(P) \}$.

The set $divergences(P)$ is the set of traces of P that lead to divergent behaviour, that is, an infinite sequence of internal events, plus all the extensions of those traces. The canonical semantics of CSP is given by the failures-divergences model; it represents a process P by the two sets $failures(P)$ and $divergences(P)$.

Here, we assume that specifications and systems are divergence free. A divergent specification is necessarily a mistake. Also, when testing, divergences raise problems of observability; generally, it is not possible to distinguish a divergent from a deadlocked system using testing. Therefore, we identify divergence with deadlock in the models of the systems under test, so that the models are divergence free; most authors circumvent the problem of observability in this way. If the system under test is divergent, the divergence is detected as a (probably forbidden) deadlock and reported as such by the verdict of the tests.

We consider two refinement relations between CSP processes: traces and failures refinement. A process P is trace-refined by a process Q, that is $P \sqsubseteq_T Q$, if, and only if, $traces(Q) \subseteq traces(P)$. For failures refinement, $P \sqsubseteq_F Q$, we require that $failures(Q) \subseteq failures(P)$, that is, Q refines P with respect to failures if, and only if, all its possible traces are traces of P, and Q may refuse a set of events and deadlock after a trace only if P may.

The first refinement relation is restrictive: it states that there are no observable behaviours of the refined process that are not behaviours of the original process. It accepts the idle process $STOP$ as a trace refinement of any other process. The second refinement relation is still restrictive on traces, but prescriptive on acceptances: it states that the refined process can only refuse to do something when the original process may refuse the same thing in the same situation. In this case, there are forbidden behaviours and mandatory behaviours.

For divergence-free processes, failures refinement as described above corresponds to the refinement relation in the canonical model of CSP: failures-divergences refinement, which is defined as follows.

$$P \sqsubseteq_{FD} Q \mathrel{\widehat=} \mathit{failures}(Q) \subseteq \mathit{failures}(P) \land \mathit{divergences}(Q) \subseteq \mathit{divergences}(P)$$

In CSP, failures refinement is defined for the stable-failures model, and not for the failures-divergences model as above. In the stable-failures model, a process is represented by its set of traces and its set of failures, that is, we have a separate record of the traces, independent of the failures. This is because, in this model, failures are only recorded for stable states, so that traces that lead to divergence are not in any of the failures. For the stable-failures model, failures refinement requires subset inclusion of both traces and failures.

In the absence of divergence, however, the failures in the stable-failures model are exactly those of the failures-divergences model, which contain a failure for all traces of the process [26, page 215]. In this case, failures inclusion, as required above in our definition of failures refinement, implies traces inclusion. In summary, for divergence-free processes, our definition of failures refinement is equivalent to the standard definition of failures refinement in CSP.

Furthermore, since the set of divergences of divergence-free processes are empty, failures-divergences refinement corresponds to failures refinement. In summary, our definition of failures refinement is the notion of refinement in the canonical model of CSP, when we are restricted to divergence-free processes.

3 Process-algebra based testing

Given a specification SP and a system under test (SUT), any testing activity is, explicitly or not, based on a satisfaction relation (also called conformance relation): $SUT\ sat\ SP$. The subject of the test is an executable system. A system is a dynamic entity. It raises tricky issues such as observability and controllability, and is sometimes submitted to peculiar physical constraints. The only way to observe it is to interact via some specific (and often limited) interface.

To test a system against a process specification, we need tests (more exactly tester processes) built on the same alphabet of events as the specification (possibly enriched by some special symbols). The execution of a given test consists in running it and the SUT in parallel.

The verdict about the success or not of a test execution depends on the observations that can be made, and it is based on the satisfaction relation. Most testing methods based on process algebras consider that two kinds of observations are possible: external events, and deadlock (that is, refusal of some external events). Deadlock is observed via time-out mechanisms: it is assumed that if the SUT does not react after a given time limit, it is blocked.

The tests are derived from the specification on the basis of the satisfaction relation, and often on the basis of some additional knowledge of the SUT and of its operational environment called *testability hypothesis*. One advantage of such a formal framework is that it makes it possible to define test sets that are unbiased and valid: they accept any SUT satisfying the specification and the testability hypothesis, and they reject any SUT that does not. Such test sets are called *exhaustive* in [12] or *complete* by other authors [3].

Exhaustive test sets are often infinite, or too large to be used in practice, but they are used as references for selecting finite, practical, test subsets according to a variety of criteria, such as additional hypotheses on the SUT [1], coverage of the specification [6, 17], or test purposes [10].

4 Testing versus refinement: testability hypotheses

Work on testing processes is traditionally based on labelled transition systems or finite state machines. To recast these results for CSP, the obvious route is to consider its operational semantics. However, there is a formal link between the operational and the denotational semantics of CSP. The main concepts usually associated with the operational semantics, like sets of initials and refusals, are also defined in terms of the denotational semantics.

This is most convenient to establish a relationship between testing and refinement. The definitions of refinement based on the denotational semantics are very simple: just subset inclusion, as presented above. It is fortunate that we are able to discuss testing notions based on the usual operational notions, but that we have a clear link to the denotational semantics. This simplifies the proofs, and allows for a formal algebraic style. When considering test criteria and test selection, though, it is necessary to refer to the operational semantics. For instance, the most popular selection criterion is transition coverage: explicit notions of states and transitions are needed to formulate it.

The refinement relations presented in Section 2.2 are natural candidates for CSP-based testing. They are, however, relations between *CSP* process, and the SUT is not a *CSP* process. A classical way of overcoming this difficulty is to assume that *the SUT behaves like some unknown CSP process*; this is our first testability hypothesis. With this assumption, we can then require that this process is a refinement of the *CSP* specification. We define that a system P behaves like a CSP process Q to mean that, in any environment, running P or running Q yields the same set of observations. If P and Q were both CSP processes, then behavioural equivalence would be characterised as refinement in both directions, but P is not a CSP process; it is a system.

In more detail, if SUT_{CSP} denotes the unknown *CSP* process that behaves as the SUT, we can consider the satisfaction relation $SP \sqsubseteq_F SUT_{CSP}$ based on failures refinement. In the sequel, we write SUT instead of SUT_{CSP} when there is no ambiguity. The use of failures refinement as a satisfaction relation has been studied in different frameworks, with various names and notations since the original definition of testing equivalence in [21]; *testing preorder*, *failure preorder*, $\leq_{te}$, *red*, *conf* are examples of the terminology that has been adopted.

The testability hypothesis requires that events of the specification abstract operations are perceived as atomic and of irrelevant duration in the SUT. It is necessary to ensure in some way that this requirement is fulfilled, for example, by developing wrappers, or performing some complementary proofs or tests.

It is interesting to note that, similarly, testing methods based on Finite State Machine (FSM) descriptions assume that the SUT behaves as an FSM with the same number of states as the specification, or a known number of states [17]. Likewise, methods based on IO-automata or IO-Transition Systems assume that the SUT behaves as an IO-automata: it is input-enabled, that is, always ready to accept any input [31]. Analogously, methods based on algebraic specifications assume that the SUT behaves as a many-sorted algebra [1].

Our second testability hypothesis is related to nondeterminism in the SUT and its influence on the verdict after test executions. This hypothesis, sometimes called the *complete testing assumption*, postulates that *there is some known integer k such that, if a test experiment is performed k times, then all possible behaviours are observed.* The issue of the number of test executions is a classical problem in black-box testing of nondeterministic systems. Its choice is generally based on empirical knowledge of the system, as, for example, its level of internal parallelism that may give rise to nondeterminism, and the length of the test. Without such hypotheses, it is hopeless to get a meaningful conclusion after a test campaign. For some hints, see [13], and for recent variants, see [15].

An example is an SUT that reads the system *clock*, and based on whether the time is an odd or an even number, performs the *even* or *odd*. Using the first testability hypothesis, we assume that it behaves like the CSP process below.

$$CReader = clock?t \rightarrow \mathbf{if}\ even(t)\ \mathbf{then}\ (even \rightarrow SKIP)\ \mathbf{else}\ (odd \rightarrow SKIP)$$

If we cannot observe the *clock*, then the behaviour of the SUT is more accurately described by the CSP process $CReader \setminus \{\!|clock|\!\}$, which, using the algebraic laws of CSP, we can show to be equal, in the failures-divergences model, to the nondeterministic process $even \rightarrow SKIP \sqcap odd \rightarrow SKIP$. Such SUT does not satisfy, according to the semantics of nondeterminism in CSP, our second testability hypothesis: no SUT that behaves like a nondeterministic CSP process does, since nondeterminism in CSP is modelled as a completely arbitrary choice, with no guarantee of balanced behaviour for any value of k. Extra knowledge of the performance of the SUT, and of the system *clock*, however, may allow us to conclude that, if the system is executed, for instance, sixty times in a single minute, it is guaranteed to read an even and an odd time at least once.

Such considerations are beyond what we aim at formalising in this paper. For the above example, for instance, we can say that $CReader \setminus \{\!|clock|\!\}$ is not an accurate model of the SUT. The extra knowledge of the performance of the SUT and of the clock means that we know that it actually behaves like a deterministic, in the sense of CSP, process whose description requires the formalisation of a notion of time, and properties of the clock. In other words, we accept the restriction imposed by the second testability hypothesis that, in the CSP sense, the SUT is deterministic, but point out that determinism may come from the observation of events that are not necessarily expected to be in the alphabet of the specification. In our example, these events are, for instance, the passage of time and the system clock.

These considerations are not relevant for works based, for example, on finite state machines, where nondeterminism is not necessarily arbitrary. In CSP, nondeterminism captures a notion of abstraction relevant for system development, and, therefore, refinement. If there is any factor that allows us to make any conclusion about the balance of nondeterministic behaviour, this means that, in the CSP sense, the system is not really nondeterministic.

5 Tests and exhaustive test sets

Failures refinement can be expressed in terms of traces refinement and a well-studied satisfaction relation [3]: the *conf* conformance relation (cf. Proposition 3.13.1 in [30]). For CSP processes P and Q, *conf* can be defined as follows.

$$\begin{array}{l} Q\ \mathit{conf}\ P \mathrel{\widehat=} \forall\, t : \mathit{traces}(P) \cap \mathit{traces}(Q) \bullet \mathit{Ref}(Q,t) \subseteq \mathit{Ref}(P,t) \\ \text{where } \mathit{Ref}(P,t) \mathrel{\widehat=} \{\, X \mid (t,X) \in \mathit{failures}(P)\,\} \end{array}$$

The above definition of $\mathit{Ref}(P,t)$ is compatible with the definition of $\mathit{refusals}(P)$ in CSP, for the process P/t [26, pages 94,197].

The following theorem establishes the relationship between failure refinement, traces refinement, and *conf*; its simple proof is in [5].

Theorem 1. $(P \sqsubseteq_F Q) \Leftrightarrow (P \sqsubseteq_T Q \wedge Q\ \mathit{conf}\ P)$

This theorem justifies the suggestion in [30] that traces refinement and *conf* can be checked separately. In this section we first study how to test a system with respect to traces refinement (that is, trace containment), and then address testing against *conf* (that is, refusal containment). These two kinds of testing are suitable for the detection of different types of faults. Testing against traces refinement makes it possible to detect forbidden behaviours, and testing against *conf* makes it possible to detect forbidden deadlocks.

5.1 Testing against traces refinement

Since trace refinement prescribes that $\mathit{traces}(\mathit{SUT}) \subseteq \mathit{traces}(\mathit{SP})$, but not the reverse, a testing strategy does not need to test that a SUT can execute the traces of SP. It is sufficient to test it against those traces in $\alpha(\mathit{SP})^{\checkmark *}$ that are not traces of SP and to check that they are refused. Moreover, it is sufficient to consider the minimal prefixes of forbidden traces that are forbidden themselves. For example, if after a trace *abc*, the event d is forbidden, then *abcda*, and *abcdb*, for example, are also forbidden, but we only need to consider *abcd*. On the other hand, if *abd* is also forbidden, we also check that it is refused.

Formally, we define a test set that proposes to the SUT the following traces.

$$\{s \frown \langle\, a \,\rangle \mid s \in \mathit{traces}(\mathit{SP}) \wedge a \notin \mathit{initials}(\mathit{SP}/s)\}$$

For one test execution, the verdict is as follows. If the trace s, followed by a deadlock is observed, then the test execution is said to be a *success*. If $s \frown a$

is observed, we have a *failure*. If a strict prefix of s followed by a deadlock is observed, then the test execution is *inconclusive*; the trace s has not been executed by the SUT, and this is acceptable for traces refinement.

As mentioned in Section 4, if the SUT is nondeterministic, then several executions of the same test must be performed and a global verdict is reached based on such a set of executions. When the SUT is known to be deterministic, there is no need for several executions of any of the tests, and the SUT passes a test as soon as the verdict is either successful or inconclusive, and not a failure.

When the SUT is nondeterministic, the following global verdict for several executions of the same test is recorded. If there is one execution with a failure verdict, the SUT does not pass the test. If for all the test executions the verdict is either success or inconclusive, the SUT passes the test.

The idea of basing the test set on the pairs (s, a) where $s \in \mathit{traces}(SP)$ and $a \notin \mathit{initials}(SP/s)$ is inspired by the work of Peleska and Siegel [24], where it is proved that based on this set of pairs, it is possible to detect all violations of traces refinement, and that this is the minimal set of pairs with this property. The notions of test and verdict that we present below, however, are slightly different, and also, we give an algebraic proof of the exhaustivity of the test set.

As mentioned in Section 3, a test execution is a run of the SUT and a test process T in parallel; we describe this in CSP as $(SUT \,[\![\, \alpha SP \,]\!]\, T) \backslash \alpha SP$. The synchronisation set, which is the interface of the system as defined in the specification, is hidden; so, the external events of SUT are internal in a test execution. This means that synchronisation between the SUT and the test proceeds immediately, and cannot be affected by the test execution environment.

On the other hand, this also means that direct observation of traces is not possible in such test executions. Thus, we introduce three special events, *pass*, *fail*, and *inc*, in the alphabet of the test in order to perform on-the-fly verdict. Using these events, we have a very direct characterisation in CSP of the verdict of a single test execution as described above.

For a finite trace $s = a_1, a_2, \ldots, a_n$ and an event a, we define a CSP test process $T_T(s, a)$ as $\mathit{inc} \rightarrow a_1 \rightarrow \ldots \mathit{inc} \rightarrow a_n \rightarrow \mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow STOP$. Formally, we can describe it as shown below.

$$\begin{array}{l}
T_T(\langle\,\rangle, a) = \mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow STOP \\
T_T(\langle \checkmark \rangle, a) = \mathit{pass} \rightarrow a \rightarrow \mathit{fail} \rightarrow STOP \\
T_T(\langle\, b \,\rangle \frown s, a) = \mathit{inc} \rightarrow b \rightarrow T_T(s, a)
\end{array}$$

Extending s with a is supposed to lead to an invalid trace; and the test aims at ruling it out. The event $\checkmark$, it if happens, is final [26, page 143], and only marks termination. So, no event should occur after it.

The execution of a test T for a given SUT, against a specification SP, is described by the CSP process below.

$$\mathit{Execution}_{SUT}^{SP}(T) = (SUT \,[\![\, \alpha SP \,]\!]\, T) \backslash \alpha SP$$

The verdict is given by the last event observed (*pass*, *fail*, or *inc*) before deadlock.

Depending on the precision required for the verdict and on some knowledge of the SUT, it is possible in some cases to use only one or two events among *pass*, *fail*, and *inc*. If the SUT is deterministic, for instance, there are no inconclusive verdicts, and *inc* can be eliminated. In fact, in our case, since we are testing for absence of forbidden traces, we can eliminate *inc* in this way. If any prefix of a tested trace is observed, the test is successful (due to the prefix closure of the sets of traces and the definition of trace refinement). Our use of the three events, however, gives a direct model of verdict for a single test execution.

We now define an exhaustive test set for trace refinement.

$$\mathit{Exhaust}_T(SP) = \{\ T_T(s, a) \mid s \in \mathit{traces}(SP) \wedge a \notin \mathit{initials}(SP/s)\ \}$$

Testing of termination is covered because it is signalled by the event ✓ that is explicitly included in the traces. As an example, we consider the process $\mathit{Counter}_2$ from Section 1. The set $\mathit{Exhaust}_T(\mathit{Counter}_2)$ contains the following tests.

$$\begin{array}{l}
T_T(\langle\,\rangle, \mathit{sub}) = \mathit{pass} \rightarrow \mathit{sub} \rightarrow \mathit{fail} \rightarrow \mathit{STOP} \\
T_T(\langle \mathit{add}, \mathit{add}\rangle, \mathit{add}) = \mathit{inc} \rightarrow \mathit{add} \rightarrow \mathit{inc} \rightarrow \mathit{add} \rightarrow \mathit{pass} \rightarrow \mathit{add} \rightarrow \mathit{fail} \rightarrow \mathit{STOP} \\
T_T(\langle \mathit{add}, \mathit{sub}\rangle, \mathit{sub}) = \mathit{inc} \rightarrow \mathit{add} \rightarrow \mathit{inc} \rightarrow \mathit{sub} \rightarrow \mathit{pass} \rightarrow \mathit{sub} \rightarrow \mathit{fail} \rightarrow \mathit{STOP} \\
T_T(\langle \mathit{add}, \mathit{add}, \mathit{sub}, \mathit{add}\rangle, \mathit{add}) = \\
\quad \mathit{inc} \rightarrow \mathit{add} \rightarrow \mathit{inc} \rightarrow \mathit{add} \rightarrow \mathit{inc} \rightarrow \mathit{sub} \rightarrow \mathit{inc} \rightarrow \mathit{add} \rightarrow \mathit{pass} \rightarrow \mathit{add} \rightarrow \mathit{fail} \rightarrow \mathit{STOP} \\
T_T(\langle \mathit{add}, \mathit{add}, \mathit{sub}, \mathit{sub}\rangle, \mathit{sub}) = \ldots
\end{array}$$

This is, of course, an infinite set since the set $\mathit{traces}(\mathit{Counter}_2)$ is infinite.

The next theorem establishes that an SUT that does not fail any of the tests in $\mathit{Exhaust}_T(SP)$ is a traces refinement of SP. Its proof can be found in [5]. For a trace t, we use $\mathit{last}(t)$ to refer to its last event.

Theorem 2 (Exhaustivity of $\mathit{Exhaust}_T$). *Given two CSP processes, SUT and SP, $SP \sqsubseteq_T SUT$ if, and only if*

$$\forall\, T_T(s, a) : \mathit{Exhaust}_T(SP);\ t : \mathit{traces}(\mathit{Execution}^{SP}_{SUT}(T_T(s, a))) \bullet \mathit{last}(t) \neq \mathit{fail}$$

As mentioned in Section 3, on the basis of this exhaustive test set, it is possible to design selection and optimisation strategies for getting finite test sets.

5.2 Testing against refusals

In this section we address the problem of testing whether an SUT behaves as a CSP process SUT_{CSP} that satisfies $SUT_{CSP}\ \mathit{conf}\ SP$. The definition of *conf* requires us to check that, after performing one of their common traces, the failures of SUT are failures of SP as well. For that we check that, after every trace s of SP, the SUT cannot refuse all events in a set A accepted by SP.

This is achieved by executing the test described by the CSP process corresponding to the trace s followed by an external choice among the events in A. Basically, a test execution proposes to the SUT the traces of the CSP process $s;\ (\Box\, a \in A \bullet a \rightarrow \mathit{STOP})$ where $s \in \mathit{traces}(SP)$ and $(s, A) \notin \mathit{failures}(SP)$. The verdict of one execution of such a test is as follows. If s, followed by a deadlock is observed, the test execution is said to be a *failure*. If a trace $s \frown a$, with $a \in A$ is observed the result of the test is said to be a *success*. If a strict prefix of s followed by a deadlock is observed, the test execution is said to be *inconclusive*; the trace s has not been executed by the SUT during this test execution.

Using the same events *inc*, *pass*, and *fail* that modelled the verdict of test executions concerned with traces refinement, we define the test for a trace $s = a_1, a_2, \ldots, a_n$ of SP and a set A such that $(s, A) \notin \mathit{failures}(SP)$, or in other words, $A \notin \mathit{Ref}(SP, s)$ as follows.

$$T_F(s, A) = \mathit{inc} \rightarrow a_1 \rightarrow \mathit{inc} \rightarrow a_2 \rightarrow \mathit{inc} \ldots a_n \rightarrow \mathit{fail} \rightarrow (\Box\, a \in A \bullet a \rightarrow \mathit{pass} \rightarrow STOP)$$

It can be formally defined as follows.

$$T_F(\langle\,\rangle, A) = \mathit{fail} \rightarrow (\Box\, a \in A \bullet a \rightarrow \mathit{pass} \rightarrow STOP)$$
$$T_F(\langle\, a\,\rangle \frown s, A) = \mathit{inc} \rightarrow a \rightarrow T_F(s, A)$$

Termination is not a special case here: every set is refused after a trace ending by $\checkmark$. If s ends in $\checkmark$, then there is no A that does not belong to the failures of SP [26, page 192], and so we do not need to define $T(\langle \checkmark \rangle, A)$.

When the SUT is nondeterministic, the global verdict for several executions of the same test is just as before. If there is one execution with a failure verdict, the SUT does not pass the test. If for all the test executions the verdict is either success or inconclusive, it passes the test. However, it is not necessary to consider all these tests because as soon as $(s, A) \notin \mathit{failures}(SP)$, then $\forall A' \supset A \bullet (s, A') \notin \mathit{failures}(SP)$. If the SUT passes the test for A, it will in principle pass the test for A'; the only concern would be with nondeterminism. On the other hand, if the SUT fails the test for A, it may pass the test for A', which offers more choices, but a problem has already been identified.

Thus for each trace s, it is sufficient to consider a subset of the sets A' such that (s, A') are not in $\mathit{failures}(SP)$; namely we consider the set $\mathcal{A}_s$ of sets $A_1, \ldots, A_m$ of events such that for all A_i, $(s, A_i) \notin \mathit{failures}(SP)$, and for all $(s, A) \notin \mathit{failures}(SP)$ there is $A_i \in \mathcal{A}_s$ such that $A \supseteq A_i$. These are the minimal acceptance sets of SP. In the case of a deadlock, there is no A such that $(s, A) \notin \mathit{failures}(SP)$, and in this case $\mathcal{A}_s$ is empty. Precisely, we propose the following exhaustive minimal test set for refusal containment.

$$\mathit{Exhaust}_{\mathit{conf}}(SP) = \{\, T_F(s, A) \mid s \in \mathit{traces}(SP) \land A \in \mathcal{A}_s \}$$

where $\mathcal{A}_s = \mathit{min}_{\subseteq}(\{\, A \mid (s, A) \notin \mathit{failures}(SP)\,\})$. For a set $\mathcal{S}$ of sets, we define $\mathit{min}_{\subseteq}(\mathcal{S}) = \{\, S \mid S \in \mathcal{S} \land \neg\, \exists\, S' \bullet S' \in \mathcal{S} \land S' \subset S\,\}$.

For the process $\mathit{Counter}_2$, we have the following minimal acceptances.

For $\langle\,\rangle$, we have $\{\, \mathit{add}\,\}$
For $\langle\, \mathit{add}\,\rangle$, we have $\{\, \mathit{add}\,\}, \{\, \mathit{sub}\,\}$
For $\langle\, \mathit{add}, \mathit{add}\,\rangle$, we have $\{\, \mathit{sub}\,\}$
For $\langle\, \mathit{add}, \mathit{sub}\,\rangle$, we have $\{\, \mathit{add}\,\}$
For $\langle\, \mathit{add}, \mathit{add}, \mathit{sub}\,\rangle$, we have $\{\, \mathit{add}\,\}, \{\, \mathit{sub}\,\} \ldots$

For each trace, there can be several refusal sets, and similarly, several sets of minimal acceptances. In the exhaustive test set, we have the following tests.

$T_F(\langle\,\rangle, \{\, add\,\}) = fail \rightarrow add \rightarrow pass \rightarrow STOP$
$T_F(\langle\, add\,\rangle, \{\, add\,\}) = inc \rightarrow add \rightarrow fail \rightarrow add \rightarrow pass \rightarrow STOP$
$T_F(\langle\, add\,\rangle, \{\, sub\,\}) = inc \rightarrow add \rightarrow fail \rightarrow sub \rightarrow pass \rightarrow STOP$
$T_F(\langle\, add, add\,\rangle, \{\, sub\,\}) = inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow fail \rightarrow sub \rightarrow pass \rightarrow STOP$
$T_F(\langle\, add, sub\,\rangle, \{\, add\,\}) = inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow fail \rightarrow add \rightarrow pass \rightarrow STOP$
$T_F(\langle\, add, add, sub\,\rangle, \{\, add\,\}) =$
$\quad inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow fail \rightarrow add \rightarrow pass \rightarrow STOP$
$T_F(\langle\, add, add, sub\,\rangle, \{\, sub\,\}) =$
$\quad inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow inc \rightarrow sub \rightarrow fail \rightarrow sub \rightarrow pass \rightarrow STOP$
...

The exhaustive test set is enough to establish conformance: an SUT for which none of the test executions deadlock after a *fail* is in conformance with SP according to *conf*; a proof is found in [5].

Theorem 3 (Exhaustivity of $Exhaust_{conf}$). *Given two CSP processes SUT and SP, $(SUT\ conf\ SP)$ if, and only if,*

$$\forall\, T_F(s, A) : Exhaust_{conf}(SP);\ (t, B) : failures(Execution^{SP}_{SUT}(T_F(s, A))) \bullet \\ last(t) \neq fail \lor B \neq \{\, inc, pass, fail\,\}$$

This theorem is similar, up to the notation, to Proposition 4.5, proved by Tretmans in [30, page 85]. The main difference there is that $initials(SP/s)$ was added to the minimal set of acceptance sets for technical reasons: the test set was defined by induction, starting from the empty trace, and all the successors of the state before the last one must appear at each induction step in order to get all the traces at the next step.

There is also a similar result by Peleska and Siegel in [24], with a rather different formulation of the sets of acceptance sets to be considered. In their theoretical work, deadlock observations are characterised in terms of maximal traces. This is not accurate for nondeterministic systems, since in the traces model a prefix of a maximal trace may be included just because it is a prefix of a possible trace, or because the system may deadlock after that prefix. It is well known that the traces model is not enough to characterise failures. In practice, one uses timeout to conclude that there is a deadlock, but in the theoretical work, this must be expressed in terms of the failures model. In the test derivation method that has been applied to several industrial systems [25], though, Peleska and Siegel handle deadlock adequately.

5.3 Running tests against systems

The two last theorems are about CSP processes, namely a specification SP, an unknown process SUT_{CSP} such that the system under test behaves like it, and some tests. They state some equivalence between, on the one hand, a conformance relation between SP and SUT_{CSP}, and, on the other hand, some properties of the traces of the parallel composition of SUT_{CSP} with the tests. These theorems are fundamental for stating practical properties on testing the SUT.

In practice, for every test in $\mathit{Exhaust}_T$ and $\mathit{Exhaust}_{conf}$, it is trivial to obtain some tester program (in any suitable language for interacting with the SUT, for instance TTCN, Concert C, ...), that behaves like the CSP test. When performing one test experiment of the SUT with the tester program corresponding to a test t, we define that $SUT\ \mathit{passes}_1\ t$ if the verdict as defined in the previous section is not a failure. Now, from the second testability hypothesis, for any test t, there is an integer k that prescribes the number of experiments that must be performed with t. We define that $SUT\ \mathit{passes}\ t$ if there is no failure observed when performing k experiments with t. We also define that $SUT\ \mathit{passes}\ TS$ when SUT passes all the tests in a test set TS.

Theorem 2 above allows to say that, under the testability hypotheses, an SUT passes all the tests of $\mathit{Exhaust}_T(SP)$ if and only if $SP \sqsubseteq_T SUT_{CSP}$. In a more mathematical form we have:

$$\mathit{Testability\ Hypotheses} \implies (SUT\ \mathit{passes}\ \mathit{Exhaust}_T(SP) \Leftrightarrow SP \sqsubseteq_T SUT_{CSP})$$

Similarly, Theorem 3 and Theorem 1 allow us to say that under the testability hypotheses, an SUT passes all the tests of $\mathit{Exhaust}_T(SP)$ and $\mathit{Exhaust}_{conf}(SP)$ if, and only if, $SP \sqsubseteq_F SUT_{CSP}$.

$$\begin{aligned}&\mathit{Testability\ Hypotheses} \implies \\ &\quad (SUT\ \mathit{passes}\ \mathit{Exhaust}_T(SP) \cup \mathit{Exhaust}_{conf}(SP) \Leftrightarrow SP \sqsubseteq_F SUT_{CSP})\end{aligned}$$

Of course the set of experiments described above, that is, the SUT passing an exhaustive test set, is not realistic since the test sets are infinite. This leads to the problem of selecting a finite subsets of these experiments. This can be done by enriching the hypotheses on the SUT with so-called selection hypotheses, keeping the same kind of properties as above: under the testability hypotheses and the selection hypotheses, the SUT passes the selected test set if and only if it is a refinement of SP. Some hints on test selection are given later in Section 7.

6 Factorisation

The tests in $\mathit{Exhaust}_T$ can be factorised, taking advantage of the fact that the set of traces of a process is prefix-closed. This factorisation decreases the number of inconclusive executions, although it may result in some adaptive tests, originally called adaptive checking sequences [17]. They were introduced for dealing with a nondeterministic SUT. Instead of submitting a preset sequence of events to the SUT, a tree of possible behaviours is submitted to it, allowing a test to follow and reveal the nondeterministic choices of the SUT.

In the case of traces refinement testing, for a trace s, we define a factorised test $\mathit{Fact}_T^{SP}(s)$ which, after each of the events of s proposes to the SUT an external choice between the continuation of the trace and those events forbidden after the prefix of the trace executed so far. It is formally defined as follows, in terms of the function $\mathit{Fact}_{tr}^{SP}(s, dt)$ that takes as an extra parameter the trace

dt that has already been executed.

$$\begin{array}{l}
\mathit{Fact}_T^{SP}(s) = \mathit{Fact}_{tr}^{SP}(s, \langle\,\rangle) \\
\mathit{Fact}_{tr}^{SP}(\langle\,\rangle, dt) = \mathit{TFail}^{SP}(dt) \\
\mathit{Fact}_{tr}^{SP}(\langle\, a\,\rangle \frown s, dt) = \mathit{TFail}^{SP}(dt) \mathrel{\square} inc \rightarrow a \rightarrow \mathit{Fact}_{tr}^{SP}(s, dt \frown \langle\, a\,\rangle) \\
\mathit{TFail}^{SP}(s) = pass \rightarrow \square\; a^f \notin \overline{initials(SP/s)} \bullet a^f \rightarrow fail \rightarrow STOP, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{provided } \overline{initials(SP/s)} \neq \emptyset \\
\mathit{TFail}^{SP}(s) = STOP, \text{ otherwise}
\end{array}$$

The function $\mathit{TFail}^{SP}(s)$ defines the CSP process that proposes all the invalid extensions of s according to SP.

For $\mathit{Counter}_2$, we have that $\mathit{Fact}_T^C(\langle add, add, sub, add\rangle)$ is the following test.

$$\begin{array}{l}
pass \rightarrow sub \rightarrow fail \rightarrow STOP \\
\square \\
inc \rightarrow add \rightarrow inc \rightarrow add \rightarrow \left(\begin{array}{l} pass \rightarrow add \rightarrow fail \rightarrow STOP \\ \square \\ inc \rightarrow sub \rightarrow inc \rightarrow add \rightarrow pass \rightarrow add \rightarrow fail \rightarrow STOP \end{array}\right)
\end{array}$$

This factorised test subsumes three tests in the set $\mathit{Exhaust}_T(\mathit{Counter}_2)$, namely, $T_T(\langle\,\rangle, sub)$, $T_T(\langle add, add\rangle, add)$, and $T_T(\langle add, add, sub, add\rangle, add)$.

The factorised tests, however, have a problem concerning coverage of behaviours, since the choices are no more under the control of the test and can be biased. For example, in the above factorised test, after the SUT performs the trace $\langle\, add, add\,\rangle$, there is a choice between the testing events *pass* and *inc* that is left up not to the test, but to the environment of the test execution. It is an external choice over events that are not in the alphabet of the SUT.

For the non-factorised tests, the environment of a test execution can be a simple process that is prepared to accept interaction on any of the events *inc*, *pass*, and *fail*, at any moment. Even in such a liberal environment, it is guaranteed that the interaction with the SUT defined by the trace that corresponds to the test is attempted in the test execution.

If the test is factorised, though, the environment of the test execution is offered choices between *pass* and *inc* events, whenever there is a possibility, according to the specification, of extending the trace or deadlocking. To ensure coverage, both choices should be tried; in a simple environment like that described above, there is no such guarantee, and more care is needed in its design.

A related concern arises when there are several possibilities of failure. By way of illustration, we consider that the alphabet of $\mathit{Counter}_2$ includes a third event *mult*, which is always refused; the definition of $\mathit{Counter}_2$ remains that presented in Section 2. In this case, the factorised test takes the following shape.

$$\begin{array}{l}
pass \rightarrow ((sub \rightarrow fail \rightarrow STOP) \mathrel{\square} (mult \rightarrow fail \rightarrow STOP)) \\
\square \\
inc \rightarrow add \rightarrow \ldots
\end{array}$$

In this case, in the beginning, if the environment of the test execution chooses to synchronise on *pass*, it then has no control over the choice between *sub* and *mult*: these events are not in its alphabet. The test is not making a choice, but rather offering it to the SUT. If the SUT can perform any of them, the mistake is going to be detected, but it is not possible to determine which of the mistaken events are possible, since the SUT is not forced to perform any of them.

If the SUT can perform one of the mistaken events, the test execution will indicate a *fail*, but it will not be clear which of the choices was made; it is a choice internal to the test execution. If the SUT can perform both *sub* and *mult* due to an internal nondeterminism, the second testability hypothesis guarantees that, after running the factorised test a certain number of times, both nondeterministic choices will be made, but again the test executions will not indicate when this has happened. If the SUT offers *sub* and *mult* in an external choice, in the test execution, the choice becomes nondeterministic, and even though the testability hypothesis guarantees that all choices will be made, there is no way, again, of identifying the reason of the problems that will be indicated.

On the other hand, if the non-factorised tests $pass \rightarrow sub \rightarrow fail \rightarrow STOP$ and $pass \rightarrow mult \rightarrow fail \rightarrow STOP$ are used instead, given the testability hypothesis, the verdict is more precise. If only one of the mistaken events can be performed by the SUT, only the test that offers that event will signal a *fail*. If the SUT can perform both *sub* and *mult* due to an internal nondeterminism, then the testability hypothesis guarantees that, after running each test a certain number of times, the nondeterministic behaviour will be revealed. Therefore, the executions for both tests will reveal a failure, and it will be observed that both mistakes are possible. If, on the other hand, the SUT can perform both *sub* and *mult* due to an external choice, then the individual non-factorised tests will make the choice and reveal the individual problems straightaway.

An even more factorised test can be defined. The automaton that defines set of traces of the so-called *canonical tester* [3] can be visualised as the tree formed by the traces of SP decorated at each node by a choice between the forbidden events after the traces from the root to the node. Such events lead to a failure verdict followed by $STOP$. In this case, there is no need for an inconclusive verdict, but the problem of coverage discussed above arises also with valid continuations of the trace. In this case, not only the environment cannot control the initial choice of mistaken events that are to be tried, that is, the choice between *sub* and *mult*, but it also cannot control whether the initial valid traces to be attempted are $\langle\, add, add\, \rangle$ or $\langle\, add, sub\, \rangle$, for example.

For this reason, the factorisation that we propose above is not the canonical tester. It provides some optimisation, in that it joins all the tests based on traces that are associated by the prefix relation, but it does not joint all the tests.

Several authors [3, 30, 24] have studied the minimisation and factorisation of the exhaustive test set for the *conf* relation. Brinksma in [3], and later on Tretmans in [30], provide some ways of building some canonical tester, that is, some nondeterministic tester process whose behaviour contains all the tests in $Exhaust_{conf}(SP)$. The factorisation is less obvious than for traces refinement since all acceptable sets must be attempted after a given trace. It leads to highly nondeterministic testers. In practice, this formulation raises issues of coverage of the test set, as discussed above for $Exhaust_T(SP)$.

7 Test Selection and Derivation

A testing strategy can be formalised as a way of selecting some finite subset of an exhaustive test set. The choice of such a strategy corresponds to stronger hypotheses on the system under test than the testability hypotheses discussed in Section 4. Such hypotheses are called selection hypotheses in [1]. Weak selection hypotheses lead to large test sets. Strong selection hypotheses lead to smaller, more practicable test sets, with the risk that they may not be fulfilled.

Various selection hypotheses can be formulated and combined depending on some knowledge of the program, some coverage criteria of the specification and ultimately cost considerations. For instance, a *regularity hypothesis* uses a size function on the tests and has the form, for a given exhaustive test set: " if the subset of the exhaustive test set made up of all the tests of size less than or equal to a given limit is passed, then the exhaustive test set is as well".

For $Exhaust_T$ and $Exhaust_{conf}$, an obvious candidate for a size function on tests such as $T_T(s, a)$ and $T_F(s, A)$ is the length of the trace s. Given the precise way in which these tests are specified in subsections 5.1 and 5.2, it is straightforward to implement test generators corresponding to these selection hypotheses. It must be noted, however, that regularity hypotheses are not always a good choice in practice. For instance, in [9], Dong and Frankl report cases where such a strategy failed to detect important faults.

A popular criterion for model-based testing (using finite state machines or labelled transition systems) is transition coverage. The corresponding selection hypothesis is a so-called *uniformity hypothesis*. Such hypotheses are common in software testing. They assume that the system behaves uniformly on some test subsets. Thus it is enough to have only one test from each of these subsets.

In the case of transition coverage, it can be reworded as: "if a subset of the exhaustive test set that exercises all the transitions is passed, then the exhaustive test set is as well". Given a finite model, it is quite feasible to develop a generator that yields a set of paths satisfying this criteria (see for instance [6, 17]). One justification of this selection hypothesis is that in model-based testing, there is a testability hypothesis similar to our first one: the SUT behaves like a finite model. Since such models have no memory, the execution of a transition is independent of the way in which it has been reached, and it is sufficient to exercise it once, checking that the output (if any) is correct, and that the arrival state in the SUT is equivalent to the arrival state in the model.

The framework that we have developed so far for CSP-based testing is based on its denotational semantics. As mentioned in Section 4, to consider selection strategies based on states or transitions, we need to work with the finite labelled transition systems derived from CSP specifications using its operational semantics. A brute-force way of transposing transition coverage into our framework could be to select one subset of $Exhaust_T$ (resp., in $Exhaust_{conf}$), such that the set of the traces s that originate the selected $T_T(s, a)$ (resp., $T_F(s, A)$) ensures the coverage of all the transitions of this labelled transition system.

However, coverage of transitions is not just exercising the transitions, but, importantly, checking that the arrival state after each transition in the SUT is a right one: it accepts or refuses events that are compatible with the conformance relation. Here the conformance relations are derived from the notions of refinement, namely, $SP \sqsubseteq_T SUT_{CSP}$ and $SUT_{CSP}\ \mathit{conf}\ SP$. These relations rely on traces, as they appear in the form of the tests, in both cases. In our testing approach, these traces correspond to full paths from the initial state, and transitions are not considered individually. Thus it is not clear that transition coverage is an adequate selection criteria for the approach of CSP-based testing that is presented here. Starting from the exhaustive test sets we have defined, new uniformity hypotheses must be investigated in order to propose pertinent selection strategies. This is the subject of future work.

8 Inputs and outputs

So far, we have considered events, with no distinction between inputs and outputs. This distinction is extremely important in testing. The choice of inputs is under the control of the tester. The outputs are under the control of the SUT and provide the information for stating the verdict.

If we consider again our small *Replicator* example given in Section 2, for example, we observe that $\mathit{Exhaust}_T(\mathit{Replicator})$ contains the following tests.

$$\begin{array}{l}
T_T(\langle c.0\rangle, c.1) = \mathit{inc} \to c.0 \to \mathit{pass} \to c.1 \to \mathit{fail} \to \mathit{STOP}\\
T_T(\langle c.0\rangle, c.2) = \mathit{inc} \to c.0 \to \mathit{pass} \to c.2 \to \mathit{fail} \to \mathit{STOP}\\
T_T(\langle c.1\rangle, c.0) = \mathit{inc} \to c.1 \to \mathit{pass} \to c.0 \to \mathit{fail} \to \mathit{STOP}\\
T_T(\langle c.0, c.0, c.0\rangle, c.1) =\\
\quad \mathit{inc} \to c.0\mathit{inc} \to c.0 \to \mathit{inc} \to c.0 \to \mathit{pass} \to c.1 \to \mathit{fail} \to \mathit{STOP}\\
\ldots
\end{array}$$

Instead of inputs or outputs, we have specific events $c.0$, $c.1$, and $c.2$. The same applies for the tests in $\mathit{Exhaust}_{\mathit{conf}}$. We proved that these sets are enough to indicate mistaken implementations, but this is under our first testability hypothesis: that the SUT can be accurately described as a CSP process. It raises an issue related to input and output that is not directly captured in CSP: that of origin of data communication.

As described in Section 2, in CSP, inputs and outputs are only syntactic sugar for synchronisations on events with composite names. For example, the process *BReplicator* defined below is not at all different from *Replicator* itself.

$$\mathit{BReplicator} = (c?x \to c!x \to \mathit{BReplicator}) \,\Box\, (c!0 \to c!0 \to \mathit{BReplicator})$$

We could say that it is a process that, instead of waiting for an input, may decide to output 0, twice. Its CSP model, however, is exactly as that of *Replicator*, since $c!0 \to c!0 \to \mathit{Replicator}$ is a choice already offered by the input.

If we are concerned with checking that an implementation does not decide to make progress without duly waiting for input from the environment, this CSP approach to modelling is not appropriate. In [26, page 302], Roscoe mentions the possibility of introducing the notion of input and output as a basic concept in CSP, by making outputs undelayable events, in much the same way that the termination event $\checkmark$ is undelayable. Much of the work on this has been carried out in the context of security and timed applications.

In particular, Schneider [27] defines "may" and "must" tests in the context of a partition of events that classifies refusable and non-refusable event, but also high-level and low-level events, for the purposes of security applications. Schneider is interested in new conformance relations, rather than refinement, and on the use of model checking, rather than testing.

An interesting line for future work is the extension of CSP to include delayable and non-delayable events as suggested in [26], and analyse how results on testing based on input-output labelled transitions systems [31], and on other similar formalisms [19], can be cast in this new version of CSP.

By taking inputs and outputs into account, we are likely to be able to factorise some of the tests that are based on the several traces generated by the implicit choice associated with a CSP input.

9 Conclusions

In this paper we have established a solid foundation for model-based testing using CSP. We have characterised the relevant testability hypotheses, and discussed their consequences. We have concentrated on testing for traces and failures refinement, but discussed the issues raised by divergence. We have proposed exhaustive test sets; the algebraic proofs of exhaustiveness are directly based on the definition of the semantics of CSP. We have considered some possible optimisations of our tests, with the observation, however, that optimisations raise issues of controllability. Finally, we have indicated the challenges imposed by inputs and outputs, and test selection based on labelled transition systems.

In [25], Peleska and Siegel present a pioneering work on CSP-based testing. They define and study two extra relations, divergence refinement and robustness. However, they point out divergent specifications are not very useful, and it is not clear how they handle divergent implementations. Their test executions do not hide the events of the specification, and so they allow interference from the environment in the interaction between the test and the SUT. Their tests use only one extra event; it characterises success. In the absence of a success, they do not reveal if the test is inconclusive or a failure. They define may and must tests; we do not make this distinction because we are able to give an inconclusive verdict. Using the three events *pass*, *inc*, and *fail*, we have a very direct model of the verdict. Their definitions of tests are based on traces, but assume the possibility of infinite traces, as described in the operational semantics. For these, their tests are not well-defined CSP processes. Since we base our work on the denotational semantics, we have a natural characterisation of the set of finite traces that are relevant for a partial, but widely accepted characterisation of CSP processes: its canonical failures-divergences model.

The failures-divergences model that we adopt here is in direct correspondence with the UTP model of CSP processes. We expect to be able to recast our results in the UTP easily, and pave the way to consider more sophisticated concurrent languages that include constructs from other programming paradigms. In particular, we are interested in a combination of CSP and Z [28] that is adequate for refinement called *Circus* [23].

The next step in our work plans, however, is the characterisation of test selection and generation techniques. We plan to use the CSP model checker to provide some empirical results on test generation and selection that will guide further work on *Circus*.

Acknowledgments

We are grateful to the Royal Society of London, who supported our collaboration through funding for a Short Visit to the UK and an International Joint Project.

References

1. G. Bernot, M.-C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A theory and a tool. *Software Engineering Journal*, 6(6):387 – 405, 1991.
2. L. Bougé, N. Choquet, L. Fribourg, and M.-C. Gaudel. Test set generation from algebraic specifications using logic programming. *Journal of Systems and Software*, 6(4):343 – 360, 1986.
3. E. Brinksma. A theory for the derivation of tests. In *Protocol Specification, testing and Verification VIII*, pages 63 – 74. North-Holland, 1988.
4. E. Brinksma and J. Tretmans. Testing Transition Systems: An Annotated Bibliography. In *MOVEP 2000 Summer School*, volume 2067 of *LNCS*, pages 187 – 195. Springer, 2001.
5. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP – Extended version. Technical report 1473, LRI, Universitè de Paris-Sud, 2007. `http://www.lri.fr/Rapports-internes/`.
6. T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE TSE*, SE-4(3):178 – 187, 1978.
7. P. Dauchy, M.-C. Gaudel, and B. Marre. Using Algebraic Specifications in Software Testing: a case study on the software of an automatic subway. *Journal of Systems and Software*, 21(3):229 – 244, 1993.
8. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Formal Methods Europe*, volume 670 of *LNCS*, pages 268 – 284. Springer, 1993.
9. R. K. Dong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM ToSEM*, 3(2):103 – 130, 1994.
10. J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An Experiment in Automatic Generation of Conformance Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29:123 – 146, 1997.

11. J. Gannon, P. McMullin, and R. Hamlet. Data abstraction implementation, specification and testing. *ACM ToPLaS*, 3(3):211–223, 1981.
12. M.-C. Gaudel. Testing can be formal, too. In *International Joint Conference, Theory And Practice of Software Development*, volume 915 of *LNCS*, pages 82 – 96. Springer, 1995.
13. M.-C. Gaudel and P. J. James. Testing algebraic data types and processes : a unifying theory. *Formal Aspects of Computing*, 10(5-6):436 – 451, 1998.
14. S. Helke, T. Neustupny, and T. Santen. Automating Test Case Generation from Z Specifications with Isabelle. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, *10th International Conference of Z Users: The Z Formal Specification Notation (ZUM 1997)*, volume 1212 of *LNCS*, pages 52 – 71. Springer, 1997.
15. R. M. Hierons. Testing from a Nondeterministic Finite State Machine Using Adaptive State Counting. *IEEE Transactions on Computers*, 53(10):1330 – 1342, 2004.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
17. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090 – 1126, 1996.
18. B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from B formal models. *The Journal of Software Testing, Verification and Reliability*, 14(2):81 – 103, 2004.
19. G. Lestiennes. *Contributions au test de logiciel basé sur des spécifications formelles*. PhD thesis, Université de Paris-Sud, 2005.
20. P. Machado and D. Sannella. Unit Testing for CASL Architectural Specifications. In *Mathematical Foundations of Computer Science*, volume 2420 of *LNCS*, pages 506 – 518. Springer, 2002.
21. R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 3(1-2):83 – 133, 1984.
22. S. Nogueira. Automatic Test Case Generation from CSP Specifications (in Portuguese). Master's thesis.
23. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 2007.
24. 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 *LNCS*, 1996.
25. J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53 – 77, 1997.
26. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
27. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. Wiley, 2000.
28. J. M. Spivey. *The Z Notation: A Reference Manual*. 2nd. Prentice-Hall, 1992.
29. T. Srivatanakul, J. A. Clark, S. Stepney, and F. Polack. Challenging formal specifications by mutation: a CSP security example. In *10th Asia-Pacific Software Engineering Conference*, pages 340 – 350. IEEE Press, 2003.
30. J. Tretmans. *A formal approach to conformance testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.
31. J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *TACAS*, volume 1055 of *LNCS*, pages 127 – 146. Springer, 1996.