Testing with inputs and outputs in CSP

Ana Cavalcanti[1] and Robert M. Hierons[2]

[1] University of York, UK
[2] Brunel University, UK

Abstract. This paper addresses refinement and testing based on CSP models, when we distinguish input and output events. From a testing perspective, there is an asymmetry: the tester (or the environment) controls the inputs, and the system under test controls the outputs. The standard models and refinement relations of CSP are, therefore, not entirely suitable for testing. Here, we adapt the CSP stable-failures model, resulting in the notion of input-output failures refinement. We compare that with the **ioco** relation often used in testing.Finally, we adapt the CSP testing theory, and show that some tests become unnecessary.

1 Introduction

As a process algebra, CSP [20] is a well established notation, with robust semantics and tools; it has been in use for more than twenty years. The availability of a powerful model checker has ensured the acceptance of CSP both in academia and industry. In the public domain, we have reports on applications in hardware and e-commerce [1, 10]. In addition, CSP has been combined with data modelling languages to cope with state-rich reactive systems [8, 16, 22, 7, 18].

Admittedly, model-based testing is not a traditional area of application for CSP. It remains the case, however, that when a CSP model is available, the possibility of using it for testing is attractive, especially in industry. In fact, a testing theory is available for CSP [4], and more recently, the use of CSP as part of testing techniques has been explored by a variety of researchers [15, 21, 5].

A difficulty, however, is the fact that CSP models do not distinguish between input and output events: they are all synchronisations. In testing, though, there is an asymmetry: the system under test (SUT) controls outputs, while the tester controls inputs. In this paper, we follow a suggestion in [19] to define a stable-failures model parameterised by sets $\mathcal{I}$ and $\mathcal{O}$ of input and output events. We call it the input-output failures model, and define input-output failures refinement.

The stable-failures model is a suitable starting point for our work because, as usual in testing, we assume that both models and systems are divergence free. In models, divergence is regarded as a mistake, and when testing an SUT, a divergence cannot be distinguished from a deadlock. On the other hand, we cater for nondeterminism in the model and in the SUT; for that, we consider failures and failures refinement, since the traces model does not capture nondeterminism.

In the software testing community, there has been much interest in input-output labelled transition systems (IOLTSs) [2], and it is typically assumed that

it is possible to observe quiescence, a state in which all enabled events are inputs and it is not possible to take an internal transition. Most approaches use the **ioco** implementation relation [23]; observations are traces that include inputs, outputs, and quiescence, the only type of refusal that can be observed.

For CSP, we define that, in the presence of inputs and outputs, a state is stable if it is not divergent, that is, stable according to the standard model, and no output is enabled. These are quiescent states, but we can observe the inputs that are enabled: models need not be input enabled. This means that a stable state in CSP is not necessarily a quiescent state in the sense adopted in IOLTS: in the standard stable-failures model, there is no notion of input, and here we do not enforce input-enabledness. We show that **ioco** and input-output failures refinement are incomparable: there are processes P and Q such that P conforms to Q under **ioco** but not under input-output failures refinement, and vice-versa. For input-enabled processes, however, **ioco** is stronger.

Other lines of work are related to ours in that they investigated refusals for inputs [11, 3, 2]. These, however, allow refusals to be observed in states from which an output is possible. The traditional explanation regarding the observation of a refusal set R is that, if the tester offers only events of R, we observe a refusal if the composition of the tester and the SUT deadlocks. Usually a tester does not block outputs from the SUT and so the composition of the SUT and the tester cannot deadlock if an output is available. As a result, we do not consider a state to be stable if an output is possible (since the SUT can change state) and so do not allow the observation of refusals in such states.

The testing theory of CSP identifies (typically infinite) test sets that are sufficient and necessary to establish (traces or failures) refinement with respect to a given CSP specification. To take advantage of knowledge about inputs and outputs, here we adapt that theory for input-output failures refinement.

In summary, we make the following contributions. First, we define input-output failures, and show how they can be calculated. The existing failures model of CSP does not cater for inputs and outputs. We also define input-output failures refinement and prove that it is incomparable with **ioco**. This relates our results to the extensive body of work on testing based on IOLTS. To obtain a refinement relation that is stronger than **ioco**, we need to use the refusal-testing model of CSP; we are exploring this issue in our current work. Finally, we adapt the CSP testing theory to input-output failures refinement, and show that some tests in the exhaustive test set for failures refinement become unnecessary.

Next, we present CSP and IOLTS. In Section 3, we present the new CSP model that considers inputs and outputs, and its refinement notion. Section 4 discusses the relationship between input-output failures refinement and **ioco**. Testing is addressed in Section 5. We conclude in Section 6.

2 Preliminaries

This section presents the notations and concepts that we use in this paper.

2.1 CSP and its stable-failures model

In CSP, the set Σ includes all events of interest. In addition, a special event $\checkmark$ is used to indicate termination, and is included in the set $\Sigma^{\checkmark} = \Sigma \cup \{\checkmark\}$. Inputs and outputs are not distinguished in Σ, and none of the CSP models caters for this distinction in controllability. We address this issue in Section 3.

The process $STOP$ represents a deadlock: a process that is not prepared to engage in any synchronisation; its only trace is the empty sequence $\langle\,\rangle$ of events. $SKIP$, on the other hand, terminates without engaging in any event: its traces are $\langle\,\rangle$ and $\langle\checkmark\rangle$. A prefixing $c \rightarrow P$ is ready to engage in a communication$c$, and then behave like the process P. A communication may be a simple event on which the process is prepared to synchronise, an input, or an output.

An external choice $P \,\Box\, Q$ offers the environment the possibility of choosing P or Q by synchronising on one of the events that they offer initially. For instance, $out.1 \rightarrow SKIP \,\Box\, out.2 \rightarrow STOP$ offers the choice to synchronise on $out.1$ (and then terminate) or $out.2$ (and deadlock). If an event is available from both P and Q, then the choice is internal: made by the process. In general $P \sqcap Q$ is the process that makes itself an internal choice to behave as P or Q.

An input communication is like $in?x$, for instance, in which a value is read through a channel in and assigned to the variable x. Also, an output $out!e$ communicates the value of the expression e through the channel out. In CSP, however, these are just modelling conventions that use events whose names are composed of a channel name and a value. For example, if the type of the channel in is T, then $in?x \rightarrow STOP$ is an abbreviation for an iterated external choice $\Box\, v : T \bullet in.v \rightarrow STOP$, where the environment is offered a choice to synchronise on any of the events $in.v$, where v is a value in T. Additionally, the output $out!1$ is just an abbreviation for the synchronisation $out.1$.

Parallelism can be described by the operator $P \,[\![\, X \,]\!]\, Q$, where P and Q are executed in parallel, synchronising on the events in the set X. For instance, in $(in?x \rightarrow out!x \rightarrow SKIP) \,[\![\, \{\!|in|\!\} \,]\!]\, (in!3 \rightarrow STOP)$, the processes synchronise on $in.3$, then $in?x \rightarrow out!x \rightarrow SKIP$ independently offers to synchronise on $out.3$, and terminates. Since $in!3 \rightarrow STOP$ deadlocks, the whole process deadlocks. The set $\{\!|in|\!\}$ contains all events $in.v$, where v is a value of the type of in.

The events on which the parallel processes synchronise are visible to the environment. To make them internal, we have the hiding operator: $P \setminus X$ is the process that behaves like P, except that occurrences of events in X are hidden.

If R is a renaming relation, that associates (old) event names to new names, the process $P[\![R]\!]$ is that obtained by renaming the events in P according to R. If an event e is related to two (or more) events in R, then every occurrence of e in P gives rise to an external choice in $P[\![R]\!]$ based on the new events.

Stable failures This semantic model of CSP characterises a process P by its set $traces(P)$ of traces and $failures(P)$ of stable failures. The latter are pairs (s, X), where s is a trace of P, after which P does not diverge, but may deadlock if only the events in the refusal set X are offered. This model distinguishes external and

internal choices (which define the same sets of traces) and can be used to reason about liveness properties (which are related to absence of deadlock).

Failures refinement $P \sqsubseteq_F Q$ of a process P by a process Q is defined as $traces(Q) \subseteq traces(P) \wedge failures(Q) \subseteq failures(P)$. So, the refined process Q can only engage in sequences of synchronisations (traces) that are possible for P, and can only deadlock when P can. Traces refinement $P \sqsubseteq_T Q$ requires only traces inclusion. It is not difficult to show that failures refinement can also be characterised as the conjunction of traces refinement and **conf**, a conformance relation used by the testing community [4]. It is defined as follows.

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

This is concerned only with traces allowed by both P and Q, but requires that after those, Q can deadlock only if P can.

2.2 CSP testing theory

A testing theory identifies testability hypotheses, notions of test and test experiment, the verdict of an experiment, and an exhaustive test set.

First of all, to reason formally about testing it is necessary to discuss formal models and thus to assume that the SUT behaves like an unknown model described using a given formalism. This is often called the *minimum hypothesis* [9]. The formalism is usually the language used for specifications. Both in [4] and here, it is assumed that the SUT behaves like an unknown CSP process.

Secondly, nondeterminism in the SUT can cause problems since whether a test leads to a failure being observed or not might depend on how nondeterminism is resolved. The standard testability hypothesis used to overcome this is that there is some known k such that the application of a test T a total of k times is guaranteed to lead to all possible responses of the SUT to T. The implications of this for testing from CSP specifications have been discussed [4].

An exhaustive test set is a (potentially infinite) set of tests that are necessary and sufficient to establish conformance with respect to a given relation [9]. The CSP testing theory identifies exhaustive test sets for traces refinement and **conf**.

Given a (specification) process P, for traces refinement, the CSP testing theory considers tests for pairs (s, a) such that s is a trace of P, but $s \frown \langle a \rangle$ is not. Given such a pair (s, a) we obtain $T_T(s, a)$ defined as follows [4].

$$T_T(\langle\rangle, a) = T_T(\langle\checkmark\rangle, a) = pass \rightarrow a \rightarrow fail \rightarrow STOP$$
$$T_T(\langle b \rangle \frown s, a) = inc \rightarrow b \rightarrow T_T(s, a)$$

We use verdict events *inc*, *pass*, and *fail*; the last of these events observed before a deadlock indicates the outcome of the test. If the trace s cannot be followed, we have an *inc*onclusive verdict. If s is executed, then we have a *pass*, but if after that the forbidden event a occurs, then we have a *fail*ure.

Execution $Execution^P_Q(T)$ of a test T for an SUT Q is described by the process $(Q \,[\![\, \alpha P \,]\!]\, T) \backslash \alpha P$. In words, the processes Q and T are executed in

parallel, synchronising on the set of events αP used in the specification, which are hidden. The set $\alpha P \subseteq \Sigma$ contains all events that P might use; its definition is a modelling decision. If the events of αP were visible, that is, not hidden, then the environment could potentially interfere with the test execution. By hiding them, we specify that they happen as soon as possible, that is, as soon as available in Q. The verdict events establish the outcome of the test execution.

Example 1. For the specification $Rep = in?x \rightarrow out!x \rightarrow Rep$, and for the empty trace, and forbidden continuation $out.0$, we have $pass \rightarrow out.0 \rightarrow fail \rightarrow STOP$ as a test for traces refinement. Similar tests arise for all output events $out.x$. With the trace $\langle in.0 \rangle$, we choose the value 0 to provide as input and have a test $inc \rightarrow in.0 \rightarrow pass \rightarrow out.1 \rightarrow fail \rightarrow STOP$. □

Example 2. A very simple traffic light controller that can be terminated at any point using an event *end* can be specified as follows.

$$\begin{aligned} Lights = {} & red \rightarrow (amber \rightarrow (green \rightarrow Lights \,\Box\, end \rightarrow SKIP) \,\Box\, end \rightarrow SKIP) \\ & \Box \\ & end \rightarrow SKIP \end{aligned}$$

Some of its tests for traces refinement are $pass \rightarrow amber \rightarrow fail \rightarrow STOP$ and $inc \rightarrow red \rightarrow inc \rightarrow amber \rightarrow inc \rightarrow green \rightarrow pass \rightarrow green \rightarrow fail \rightarrow STOP$. □

Using T_T, we obtain the following exhaustive test for traces refinement [4].

$$Exhaust_T(P) = \{\, T_T(s, a) \mid s \in traces(P) \wedge a \notin initials(P/s) \}$$

The process P/s characterises the behaviour of P after engaging in the trace s, and $initials(P)$ gives the set of events initially available for interaction with P.

As defined above, Q **conf** P requires checking that after a trace of both P and Q, the refusals of Q are refusals of P as well. For that, we check that after a trace of P, Q cannot refuse all events in a minimal acceptance set A of P. An acceptance set A is such that (s, A) is not a failure of P; it is minimal if it has no acceptance set as a proper subset. Formally, testing for **conf** is performed by proposing to Q the traces s in $traces(P)$, and then an external choice over the events a in a minimal acceptance set of P. For a trace s and a (minimal) acceptance set A, the test process $T_F(s, A)$ is defined as follows.

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

As for traces-refinement tests, the last event before a deadlock gives the verdict.

The exhaustive test set for conformance to P is shown below; it contains all $T_F(s, A)$ formed from traces $s \in traces(P)$, and minimal acceptance sets A [4].

$$Exhaust_{conf}(P) = \{\, T_F(s, A) \mid s \in traces(P) \wedge A \in \mathcal{A}_s \}$$

The set $\mathcal{A}_s = min_{\subseteq}(\{\, A \mid (s, A) \notin failures(P) \,\})$ contains the minimal acceptances after s. As already indicated, for failures refinement, the exhaustive test set is $Exhaust_T(P) \cup Exhaust_{conf}(P)$, covering traces refinement and **conf**.

2.3 Input-output labelled transition systems

An input-output labelled transition system (IOLTS) is a labelled transition system in which we distinguish between inputs and outputs. IOLTSs have received much attention in the software testing literature, due to the asymmetry between input and outputs in testing. Formally, IOLTSs can be defined as follows.

Definition 1. *An* input-output labelled transition system *is defined by a tuple* $P = (\mathcal{P}, \mathcal{I}, \mathcal{O}, \mathcal{T}, p_{in})$ *in which* $\mathcal{P}$ *is a countable set of states,* $p_{in} \in \mathcal{P}$ *is the initial state,* $\mathcal{I}$ *is a countable set of inputs,* $\mathcal{O}$ *is a countable set of outputs, and* $\mathcal{T} \subseteq \mathcal{P} \times (\mathcal{I} \cup \mathcal{O} \cup \{\tau\}) \times \mathcal{P}$*, where* τ *represents an internal action, is the transition relation. The sets* $\mathcal{I}$ *and* $\mathcal{O}$ *are required to be disjoint and* $\tau \notin \mathcal{I} \cup \mathcal{O}$*. A transition* (p, a, p') *means that from* p *it is possible to move to* p' *with action* $a \in \mathcal{I} \cup \mathcal{O} \cup \{\tau\}$*. A state* $p \in \mathcal{P}$ *is* quiescent *if all transitions from* p *are labelled with inputs. We represent quiescence by* δ *and extend* $\mathcal{T}$ *by adding the transition* (p, δ, p) *for each quiescent* p*, calling the resulting relation* $\mathcal{T}_\delta$*. Further,* P *is* input enabled *if for all* $p \in \mathcal{P}$ *and* $?i \in \mathcal{I}$ *there is some* $p' \in \mathcal{P}$ *such that* $(p, ?i, p') \in \mathcal{T}$*.*

A sequence $s = a_1, \ldots, a_n \in (\mathcal{I} \cup \mathcal{O} \cup \{\delta\})^*$ of actions is a trace of P if there exists a sequence $(p_1, a_1, p_2), (p_3, a_2, p_4), \ldots, (p_n, a_n, p_{n+1})$ of transitions in $\mathcal{T}_\delta$ such that P can move from p_{in} to p_1 through a sequence of internal transitions (those with action τ) and for all $1 \leq i \leq n$ it is possible to move from p_{2i} to p_{2i+1} through internal transitions. Given an IOLTS P, we let $tr^{\mathbf{io}}(P)$ denote the set of traces of P. Roughly speaking, an IOLTS action corresponds to a CSP event, except that δ is not a CSP event, and $\checkmark$ is not an action in IOLTSs.

3 Failures with inputs and outputs

The traces of a process P are not affected by the controllability of the events. Therefore, the distinction between inputs and outputs does not affect the trace model of CSP. Controllability, however, affects the notion of failures.

We define the input-output stable failures $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)$ of a process in a context where the disjoint sets $\mathcal{I}$ and $\mathcal{O}$ of events identify the inputs and the outputs. Synchronisation events, which do require agreement from the environment, are regarded as inputs. Stability here is characterised by the unavailability of outputs (as well as internal events). A process that is ready to output is not in a stable state because we assume that outputs are under the sole control of the process: they do not require synchronisation, and therefore cannot be refused by the environment. A process that can output can, therefore, choose to output and evolve to a new state before considering any interaction with the environment.

It is our assumption that the SUT need not be input enabled, but we implicitly require the environment to be input enabled, since it cannot block outputs from the SUT. It is clear that many systems are not input enabled since, for example, they provide interfaces where certain fields or buttons may not be available depending on the state. (Such an SUT might be regarded as input enabled if we consider inputs at the level of events such as mouse clicks, but this

level of abstraction is rarely suitable for modelling.) Such an SUT normally does not provide the user with the option to refuse outputs, since it controls the user interface: to block outputs the user has to close down the interface, a process that may send an input to the SUT in any case.

Our definition of $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)$ in terms of the set $\mathit{failures}(P)$ of failures of P is as follows. (This definition is similar to that of a hiding $P \setminus \mathcal{O}$, but the output events are not removed from the trace.)

Definition 2. $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P) = \{\, (s, X) \mid (s, X \cup \mathcal{O}) \in \mathit{failures}(P) \}$

As already said, the stable states are those in which the output events are not available: those in which P can refuse all of them. They are characterised by a failure $(s, X \cup \mathcal{O})$. For each of them, we keep in $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)$ the failure (s, X). Since refusals are downward closed, (s, X) is also a failure of P. By considering just the failures for which $(s, X \cup \mathcal{O}) \in \mathit{failures}(P)$, we keep in $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)$ just the failures of P in its stable states. For every process P and disjoint sets $\mathcal{I}$ and $\mathcal{O}$, the pair $(\mathit{traces}(P), \mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P))$ satisfies all the healthiness conditions of the stable-failures model [20]. Proof of this and all other results presented here can be found in an extended version of this paper [6].

Example 3. We present the input-output failures of the process $E3$ below, in the context indicated in its definition; the events corresponding to communications over the channels inA and inB are inputs, and those over outA and outB are outputs. We have inputs in choice and an input in choice with an output.

$$E3 = \left(\mathit{inA}?x \to \mathit{STOP} \,\Box\, \mathit{inB}?x \to \left(\begin{array}{l} \mathit{inA}?x \to \mathit{outA}!1 \to \mathit{STOP} \\ \Box \\ \mathit{outB}!1 \to \mathit{outA}!1 \to \mathit{STOP} \end{array} \right) \right)$$

For conciseness, we omit below the parameter $(\{\!|\mathit{inA}, \mathit{inB}|\!\}, \{\!|\mathit{outA}, \mathit{outB}|\!\})$ of $\mathit{IOfailures}$. Also, if the value x communicated in an event $c.x$ does not matter, we write $c?x$ in failures. For example, $(\langle \mathit{inA}?x \rangle, \{\!|\mathit{inA}, \mathit{inB}, \mathit{outA}, \mathit{outB}, \checkmark|\!\})$ represents a set of failures: one for each of the possible values of x in $\mathit{inA}.x$.

$$\begin{array}{l} \mathit{IOfailures}(E3) = \{\, (\langle\rangle, \{\!|\mathit{outA}, \mathit{outB}, \checkmark|\!\}), \ldots \\ \qquad\qquad (\langle \mathit{inA}?x \rangle, \{\!|\mathit{inA}, \mathit{inB}, \mathit{outA}, \mathit{outB}, \checkmark|\!\}), \ldots, \\ \qquad\qquad (\langle \mathit{inB}?x, \mathit{inA}?x, \mathit{outA}.1 \rangle, \{\!|\mathit{inA}, \mathit{inB}, \mathit{outA}, \mathit{outB}, \checkmark|\!\}), \ldots, \\ \qquad\qquad (\langle \mathit{inB}?x, \mathit{outB}.1, \mathit{outA}.1 \rangle, \{\!|\mathit{inA}, \mathit{inB}, \mathit{outA}, \mathit{outB}, \checkmark|\!\}, \ldots\} \end{array}$$

In the above description, we omit the failures that are obviously included due to downward closure of refusals. For instance, the empty trace $\langle\,\rangle$ is paired with all subsets of $\{\!|\mathit{outA}, \mathit{outB}, \checkmark|\!\}$ in $\mathit{IOfailures}(E3)$. We observe, however, that there are no failures for traces $\langle \mathit{inB}?x \rangle$ or for traces $\langle \mathit{inB}?x, \mathit{inA}?x \rangle$ and $\langle \mathit{inB}?x, \mathit{outB}.1 \rangle$, for any values of x. This is because, after each of them, an output is available. So, the states after these traces are not stable. □

Example 4. For outputs in choice, we have the example below.

$$E4 = \mathit{out}!0 \to \mathit{inA}?x \to \mathit{STOP} \,\Box\, \mathit{out}!1 \to \mathit{inB}?x \to \mathit{STOP}$$

$$\begin{aligned} \mathit{IOfailures}(E4) = \{\, & (\langle \mathit{out}.0 \rangle, \{\!|\, \mathit{out}, \mathit{inB}, \checkmark \,|\!\}), \ldots, (\langle \mathit{out}.1 \rangle, \{\!|\, \mathit{out}, \mathit{inA}, \checkmark \,|\!\}), \ldots, \\ & (\langle \mathit{out}.0, \mathit{inA}?x \rangle, \{\!|\, \mathit{out}, \mathit{inA}, \mathit{inB}, \checkmark \,|\!\}), \ldots, \\ & (\langle \mathit{out}.1, \mathit{inB}?x \rangle, \{\!|\, \mathit{out}, \mathit{inB}, \mathit{inB}, \checkmark \,|\!\}), \ldots\} \end{aligned}$$

There are no failures for $\langle\rangle$, since outputs are immediately available. □

Input-output failures cannot distinguish between internal and external choice of outputs. In the example above, for instance, the failures would not change if we had an internal choice. This reflects the fact that, in reality, the environment cannot interfere with outputs. As shown by the next example, however, input-output failures can distinguish internal and external choice in other situations.

Example 5. We have a nondeterministic choice between an input and an output.

$$E5 = \mathit{inp}?x \to \mathit{STOP} \sqcap \mathit{out}!1 \to \mathit{STOP}$$

Corresponding to the possibility of the (internal) choice of $\mathit{inp}?x \to \mathit{STOP}$, we have failures for $\langle\rangle$. They indicate that an input cannot be refused in this case.

$$\begin{aligned} \mathit{IOfailures}(E5) = \{\, & (\langle\,\rangle, \{\!|\, \mathit{out}, \checkmark \,|\!\}), \ldots, \\ & (\langle \mathit{inp}?x \rangle, \{\!|\, \mathit{inp}, \mathit{out}, \checkmark \,|\!\}), \ldots, (\langle \mathit{out}.1, \rangle, \{\!|\, \mathit{inp}, \mathit{out}, \checkmark \,|\!\}), \ldots\} \end{aligned}$$

If we were to use an external choice in $E5$, then its initial state would be unstable, as $\mathit{out}.1$ would be possible, and there would be no failure for $\langle\rangle$. □

Calculating input-output failures As illustrated, not all traces are included in an input-output failure. This is also the case in the standard stable-failures model, where missing traces are those that lead to a divergent state. Here, missing traces lead to a state where either divergence or an output is possible.

Using Definition 2, we can calculate characterisations of input-output failures for the various CSP processes. A summary is provided in Table 1; proof of all these results is provided in [6]. To allow us to consider $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)$ as characterising a semantics for CSP processes P with inputs and outputs, we need to define some well formedness rules. First, as already indicated, the sets $\mathcal{I}$ and $\mathcal{O}$ of inputs and outputs must form a partition of Σ.

Second, in a parallelism $P \,[\![\, X \,]\!]\, Q$, the processes P and Q must have the same inputs and outputs $\mathcal{I}$ and $\mathcal{O}$, and X can only contain inputs ($X \subseteq \mathcal{I}$). If X contains an output, unstable states of P and Q in which outputs are available may become deadlocked, and so stable, if P and Q cannot agree on the output. This is a strong restriction, but we observe that it is necessary only for the compositional calculation of input-output failures (and consequently for the use of $\mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)$ as a semantic function). It is always possible to use Definition 2 to calculate input-output failures directly.

Third, for renaming, we require that the controllability of events is not changed: renamed inputs are still inputs, and similarly, renamed outputs are still outputs. In this way, given the sets $\mathcal{I}$ and $\mathcal{O}$ of inputs and outputs of $P[\![R]\!]$, we characterise its input-output failures in terms of those for P when its input and outputs are the relational images of $\mathcal{I}$ and $\mathcal{O}$ under R^{-1}, the inverse of R.

Process P	$IOfailures^{(\mathcal{I},\mathcal{O})}(P)$
$STOP$	$\{ (\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark} \}$
$SKIP$	$\{ (\langle\rangle, X) \mid X \subseteq \Sigma \} \cup \{ (\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark} \}$
$a \rightarrow P$	$\{(\langle\rangle, X) \mid a \notin \mathcal{O} \wedge a \notin X\} \cup \{(\langle a\rangle \frown s, X) \mid (s, X) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P)\}$
$P \sqcap Q$	$IOfailures^{(\mathcal{I},\mathcal{O})}(P) \cup IOfailures^{(\mathcal{I},\mathcal{O})}(Q)$
$P \,\Box\, Q$	$\{(\langle\rangle, X) \mid (\langle\rangle, X) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P) \cap IOfailures^{(\mathcal{I},\mathcal{O})}(Q)\} \cup$ $\{(s, X) \mid (s, X) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P) \cup IOfailures^{(\mathcal{I},\mathcal{O})}(Q) \wedge s \neq \langle\rangle\} \cup$ $\{(\langle\rangle, X) \mid X \subseteq \Sigma \wedge \langle\checkmark\rangle \in traces(P) \cup traces(Q)\}$
$P \,[\![X]\!]\, Q$	$\{(u, R) \mid \exists\, Y, Z \bullet$ $Y \cup Z \cup \mathcal{O} = R \cup \mathcal{O} \wedge (Y \setminus (X \cup \{\checkmark\})) \cup \mathcal{O} = (Z \setminus (X \cup \{\checkmark\})) \cup \mathcal{O} \wedge$ $\exists\, s, t \bullet (s, Y) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P) \wedge (t, Z) \in IOfailures^{(\mathcal{I},\mathcal{O})}(Q) \wedge$ $u \in s \,[\![X]\!]\, t\}$
$P \setminus X$	$\{ (s \setminus X, Y) \mid (s, Y \cup X) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P) \}$
$P;\ Q$	$\{(s, X) \mid s \in \Sigma^* \wedge (s, X \cup \mathcal{O} \cup \{\checkmark\}) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P) \} \cup$ $\{(t \frown u, X) \mid t \frown \langle\checkmark\rangle \in traces(P) \wedge (u, X) \in IOfailures^{(\mathcal{I},\mathcal{O})}(Q)\}$
$P[\![R]\!]$	$\{ (s', X) \mid \exists\, s \ R \ s' \wedge (s, R^{-1}(X)) \in IOfailures^{(R^{-1}(\mathcal{I}), R^{-1}(\mathcal{O}))}(P) \}$

Table 1. $IOfailures^{(\mathcal{I},\mathcal{O})}(P)$ semantics

The input-output failures of $STOP$ and $SKIP$ are the same as their standard failures. For $a \rightarrow P$, the characterisation is slightly different. Before a takes place, it cannot be refused, but this is a stable state only if a is not an output. So, we only include failures for $\langle\rangle$ if a is not an output. For internal and external choices, parallelism, hiding and sequence, input-output failures can be calculated in much the same way as standard failures. In the definition of $IOfailures^{(\mathcal{I},\mathcal{O})}(P[\![R]\!])$, we write $R^{-1}(X)$ for the relational image of the set X under R.

Input-output failures refinement Having introduced a new notion of failure, we can now introduce the corresponding definition of refinement.

Definition 3 (Input-output Failures Refinement).

$$P \sqsubseteq_{IOF}^{(\mathcal{I},\mathcal{O})} Q \mathrel{\widehat{=}} traces(Q) \subseteq traces(P) \wedge IOfailures^{(\mathcal{I},\mathcal{O})}(Q) \subseteq IOfailures^{(\mathcal{I},\mathcal{O})}(P)$$

This is a straightforward adaptation of the notion of failures refinement.

Chaos is the bottom of this relation (as well as of the standard failures-refinement relation). This is the process that can nondeterministically choose to deadlock, accept or reject any of the inputs, and produce any of the outputs. Its set of failures includes all possible failures, and consequently, so does its set of input-output failures: $(s, X \cup \mathcal{O}) \in failures(Chaos)$ for every s and X. Like in the standard model, the top of the refinement relation is **div**, the process that diverges immediately. Its set of (input-output) failures is empty, independently

of which events are inputs and which are outputs. Recursion is handled as in the standard failures model: as the least fixed point with respect to $\subseteq$.

Reduction of nondeterminism and possible deadlocks is a way of achieving input-output failures refinement. For example, we can refine the process $E5$ in Example 5 to either $inp?x \rightarrow STOP$ or $out!1 \rightarrow STOP$. We have the following.

Lemma 1. *$P \sqcap Q \sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF} P$ and $P \sqcap Q \sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF} P \,\Box\, Q$*

This follows directly from the definitions; all proofs omitted here are in [6].

4 Input-output failures refinement and ioco

As already mentioned, much of the work on testing is based on labelled-transition systems, and to cater for inputs and outputs, IOLTSs have been widely explored. In this context, the implementation relation **ioco** [23] is normally adopted. In the context of CSP, on the other hand, the conformance relation is refinement, and in the previous section we introduced input-output failures refinement ($\sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF}$). In this section, we explore the relationship between **ioco** and $\sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF}$.

First of all, we provide a definition of **ioco**. We use two functions: given a state q and a trace $s \in (\Sigma \cup \{\delta\})^*$, q **after** s is the set of states reachable from q using s. Furthermore, we have that $\mathbf{out}(q)$ is the set of $a \in (\mathcal{O} \cup \{\delta\})$ such that, from q, the next observable event could be a. This definition extends to sets of states in the usual way: for a set $\mathcal{P}'$ of states we have that $\mathbf{out}(\mathcal{P}') = \cup_{q \in \mathcal{P}'} \mathbf{out}(q)$.

Definition 4. *If Q is input enabled, we say that Q conforms to P under* **ioco**, *written Q **ioco** P, if $\mathbf{out}(Q \textbf{ after } s) \subseteq \mathbf{out}(P \textbf{ after } s)$, for every $s \in tr^{\mathbf{io}}(P)$.*

As a simplifying assumption **ioco** requires implementations to be input enabled, which is natural for some domains of application. This avoids, for example, accepting an implementation that can initially either deadlock or behave like P as a valid implementation of P (a feature, for instance, of the CSP traces model).

Input-enabled processes cannot have (reachable) termination states. This indicates that **ioco** does not distinguish termination from deadlock, but that is not all. Conformance under **ioco** does not guarantee refinement.

Theorem 1 *There are Q and P such that Q **ioco** P, but not $P \sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF} Q$.*

The observation of input-output failures can provide additional observational power, when compared to traces that include quiescence. For example, it is possible to distinguish internal and external choice of inputs. So, it is no surprise that Q **ioco** P does not imply that $P \sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF} Q$. Proofs of the above theorem and of all other theorems in the sequel can be found in [6].

On the other hand, under **ioco** it is possible to observe the failure to produce output (quiescence) before the end of a trace, while under input-output failures refinements we only observe refusal sets at the end of a trace.

Theorem 2 *There are Q and P such that $P \sqsubseteq^{(\mathcal{I},\mathcal{O})}_{IOF} Q$, but not Q **ioco** P.*

We observe that the above results are not specific to input-output failures.

In summary, $\sqsubseteq_{IOF}^{(\mathcal{I},\mathcal{O})}$ and **ioco** are generally incomparable. On the other hand, if we consider only input-enabled processes then **ioco** is strictly stronger.

Theorem 3 *If Q and P are input enabled and Q* **ioco** *P, then $P \sqsubseteq_{IOF}^{(\mathcal{I},\mathcal{O})} Q$. It is possible, however, that $P \sqsubseteq_{IOF}^{(\mathcal{I},\mathcal{O})} Q$, but not Q* **ioco** *P.*

These results do not reflect on the value of one or the other conformance relation. In the context of CSP, refinement, rather than **ioco** is the natural notion of conformance, and input-enabledness is not an adopted assumption, although it is possible to define input-enabled processes in CSP.

5 Testing

This section explores testing; we adapt the work developed for stable failures refinement [4] described in Section 2. As already said, the notion of a trace is not affected by the distinction between inputs and outputs. We can therefore reuse the previous approach for testing for traces refinement. Additionally, like in [4], we define the relation $\mathbf{conf}^{\mathcal{O}}$ that models the requirements, under $\sqsubseteq_{IOF}^{(\mathcal{I},\mathcal{O})}$, on the input-output failures. As expected, it is similar to **conf**.

$$Q\ \mathbf{conf}^{\mathcal{O}}\ P \mathrel{\widehat{=}} \forall\, t : \mathit{traces}(P) \cap \mathit{traces}(Q) \bullet \mathit{Ref}^{\mathcal{O}}(Q,t) \subseteq \mathit{Ref}^{\mathcal{O}}(P,t)$$
$$\text{where } \mathit{Ref}^{\mathcal{O}}(P,t) \mathrel{\widehat{=}} \{X \mid (t,X) \in \mathit{IOfailures}^{(\mathcal{I},\mathcal{O})}(P)\}$$

The following shows the relevance of $\mathbf{conf}^{\mathcal{O}}$.

Lemma 2. $P \sqsubseteq_{IOF}^{(\mathcal{I},\mathcal{O})} Q \Leftrightarrow \mathit{traces}(Q) \subseteq_T \mathit{traces}(P)\ \wedge\ Q\ \mathbf{conf}^{\mathcal{O}}\ P$

The proof is similar to that in [4] for $\sqsubseteq_T$ and **conf**, and is in [6].

Since $\mathit{Exhaust}_T(P)$ is exhaustive with respect to traces refinement, it is sufficient to show how an exhaustive test can be produced for $\mathbf{conf}^{\mathcal{O}}$. Like for **conf**, by definition, to check $Q\ \mathbf{conf}^{\mathcal{O}}\ P$ it is sufficient to check the refusal sets in states reached by traces in $\mathit{traces}(P) \cap \mathit{traces}(Q)$. Since Q is not known, we introduce tests to check the refusal sets after traces of P.

We use an approach similar to that of [4], which is formalised in the definition of T_F as presented in Section 2.2. At the end of a trace of P, we give a verdict *fail*, but propose a choice of events which, if accepted by Q, lead to a *pass* verdict. In the case of $\mathbf{conf}^{\mathcal{O}}$, however, we observe that if a trace leads to a state of P that is unstable because it may produce an output, then a potentially non-conformant implementation might deadlock, produce an unexpected output, or move to another stable state before producing an output. Deadlock is not allowed, and our tests for $\mathbf{conf}^{\mathcal{O}}$ check that. Unexpected outputs are checked by the tests for traces refinement. Finally, moving to another stable state may or may not be allowed (due to the presence of nondeterminism), and whether the inputs then required are allowed or not is also checked by traces refinement. We, therefore, do not need as many tests for $\mathbf{conf}^{\mathcal{O}}$ as we needed for **conf**.

Example 6. Consider the following example in which there is an internal choice between an input and two possible outputs.

$$E8 = inp?x \rightarrow STOP \sqcap (out!1 \rightarrow STOP \,\Box\, out!2 \rightarrow STOP)$$

Under stable failures the maximal refusal sets after $\langle\rangle$ are $\{out.1, out.2, \checkmark\}$ and $\{\!|inp, \checkmark|\!\}$, and so the minimal acceptances $\mathcal{A}_{\langle\rangle}$ contains all sets that contain $out.1$ and one input and all sets that contain $out.2$ and one input. For each of these, we have one test for **conf**. On the other hand, in the initial internal choice, only the choice of $inp?x$ corresponds to a stable state and in this state only outputs are refused. In this case, as formalised below, the minimal acceptances is the set of sets that contain only one input. So, we have fewer tests for $\mathbf{conf}^{\mathcal{O}}$. □

Formally, like for $Exhaust_{conf}$, we consider pairs $(s, A) \notin IOfailures^{(\mathcal{I},\mathcal{O})}(P)$. We, however, restrict ourselves to $A \subseteq \mathcal{I}$. This is justified by the following lemma.

Lemma 3. *For every P with output events $\mathcal{O}$, set of events Y such that $Y \subseteq \mathcal{O}$, and $(s, X) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P)$, we have $(s, X \cup Y) \in IOfailures^{(\mathcal{I},\mathcal{O})}(P)$.*

The proof of this lemma is in [6]. Due to its converse $(s, A \cup Y) \notin IOfailures^{(\mathcal{I},\mathcal{O})}(P)$ implies $(s, A) \notin IOfailures^{(\mathcal{I},\mathcal{O})}(P)$. Therefore, since for the construction of tests we are interested in minimal acceptances, it is enough to consider $A \subseteq \mathcal{I}$. We check that such an A is not a refusal set of a *stable state* of Q reached by s.

We check this by using a test based on s followed by an external choice of the events in $A \cup \mathcal{O}$. The set $\mathcal{O}$ is included to ensure that we get verdict *fail* only through the observation of a refusal of A in a stable state of Q, in which outputs are not available. If the test deadlocks, then this means that (s, A) is in $IOfailures^{(\mathcal{I},\mathcal{O})}(Q)$ and so we return verdict *fail*. If an event from $A \cup \mathcal{O}$ occurs after s then we return verdict *pass*. In fact, if an output is produced, the state reached by s was not stable, and an *inc* verdict would also be appropriate, but this distinction is not necessary: what we want to ensure is that a deadlock is not possible. Finally, if s is not followed, then the verdict is *inc*. This leads to the test $T_F(s, A \cup \mathcal{O})$, using the previously defined function T_F.

In conclusion, we obtain the following test set for input-output failures.

$$Exhaust^{\mathcal{O}}_{conf}(P) = \{\, T_F(s, A \cup \mathcal{O}) \mid s \in traces(P) \wedge A \in A^{\mathcal{O}}_s(P)\}$$
$$A^{\mathcal{O}}_s(P) = min\{A \subseteq \mathcal{I} \mid (s, A) \notin IOfailures^{(\mathcal{I},\mathcal{O})}(P)\}$$

This test is exhaustive for $\mathbf{conf}^{\mathcal{O}}$.

Example 7. We consider $E8$ again. In $IOfailures^{(\mathcal{I},\mathcal{O})}(E8)$, the failures with trace $\langle\rangle$ have subsets of $\{out.1, out.2, \checkmark\}$ as refusals, and so $A^{\mathcal{O}}_{\langle\rangle}(E8)$ is the set of sets that contain only one input as already said. For each of these, we have one test that also accepts all outputs. For example, $T_F(\langle\rangle, \{inp.1\} \cup \mathcal{O})$. □

Theorem 4 *For Q and P such that $traces(Q) \subseteq traces(P)$, Q $\mathbf{conf}^{\mathcal{O}}$ P if, and only, if there is no $T_F(s, A \cup \mathcal{O}) \in Exhaust^{\mathcal{O}}_{conf}(P)$ such that Q fails $T_F(s, A \cup \mathcal{O})$.*

It might appear that $Exhaust^{\mathcal{O}}_{conf}(P)$ does not check refusals after traces s that cannot take P to a stable state. This is not the case: since s cannot reach a stable state of P, for all $A \subseteq \mathcal{I}$ we have $(s, A) \notin IOfailures^{(\mathcal{I},\mathcal{O})}(P)$ and so $A^{\mathcal{O}}_s(P) = min\{A \subseteq \mathcal{I} \mid (s, A) \notin \varnothing\} = min\{A \subseteq \mathcal{I}\}$. Thus, $A^{\mathcal{O}}_s(P) = \{\varnothing\}$. As required, the SUT Q fails the corresponding test $T_F(s, \mathcal{O})$ if s can reach a stable state of Q, in which case $Ref(Q, s)$ is non-empty despite $Ref(P, s)$ being empty. The other special case is where s can reach a deadlock or terminating state of P, in which case $\{A \subseteq \mathcal{I} \mid (s, A) \notin IOfailures^{(\mathcal{I},\mathcal{O})}(P)\}$ is empty and so $A^{\mathcal{O}}_s(P)$ is empty. We therefore obtain no tests for input-output failures with s; this is what we expect since all refusal sets are allowed after s.

We now consider how $Exhaust^{\mathcal{O}}_{conf}(P)$ relates to the set $Exhaust_{conf}(P)$ for testing for stable-failures refinement [4]. In the case of $E8$, we have already noted that there are twice as many tests in $Exhaust_{conf}(E8)$. The traffic light example is more extreme since under input-output failures the only stable states are terminating states. Thus, $Exhaust^{\mathcal{O}}_{conf}(Lights) = \{T_F(\langle\rangle, \mathcal{O}), T_F(\langle red\rangle, \mathcal{O}), T_F(\langle red, amber\rangle, \mathcal{O}), \ldots\}$: these tests check that the SUT cannot deadlock or terminate without first receiving *end*. In contrast, $Exhaust_{conf}(Lights)$ would also include tests such as $T_F(\langle red\rangle, \{end\})$ and $T_F(\langle red\rangle, \{amber\})$, which check that after *red* the process cannot refuse *amber* and also cannot refuse *end*. The following shows how the sets $A_s(P)$ and $A^{\mathcal{O}}_s(P)$ relate, the proof being in [6].

Lemma 4. *If* $(s, A) \in A^{\mathcal{O}}_s(P)$ *then there exists* $Y \subseteq \mathcal{O}$ *where* $(s, A \cup Y) \in A_s(P)$.

Thus, for every test produced for input-output failures refinement there is a corresponding test produced for stable failures refinement. This shows that $Exhaust^{\mathcal{O}}_{conf}(P)$ contains no more tests than $Exhaust_{conf}(P)$. As we have seen with $E6$ and *Lights*, $Exhaust^{\mathcal{O}}_{conf}(P)$ can contain fewer tests.

6 Conclusions

This paper has explored a model, a refinement relation, and a testing theory for CSP where we distinguish between inputs and outputs. This distinction is important for testing since the tester (that is, the environment) controls inputs and the SUT controls outputs. It is normal to assume that the environment does not block outputs and, as a result, the composition of a tester and the SUT can only deadlock if the SUT is in a stable state where outputs are not available. We have thus defined a notion of failures, called input-output stable failures, which distinguish between inputs and outputs and only allow refusals to be observed in stable states where no outputs are enabled. We have defined the notion of input-output failures, showed how these can be calculated for (well-formed) CSP processes and defined a corresponding notion of refinement. We have also showed how this relates to **ioco** and adapted the CSP testing approach of [4].

Refusals in the presence of inputs have been studied in [11, 3, 2]. The key difference between these previous approaches and ours is that they allowed refusals to be observed in states where outputs are enabled. One possible justification for this approach is the symmetry between the SUT and the tester, neither of which

need to be input enabled; such a tester can block outputs and so can observe refusals in states where outputs are enabled. What we suggest is that there are systems that are not input enabled but whose environment is input-enabled. It seems likely that there will be classes of system for which we can observe refusals in states in which outputs are enabled, and so we can use implementation relations previously defined for IOLTSs, but also classes of systems for which these previous approaches are not suitable. For example, synchronous devices constitute environments that are not input enabled, on the other hand, as previously discussed, systems that write to the screen (or to any asynchronous device) and basically control its interface have an environment that is input enabled.

In occam, a programming language based on CSP [14], for instance, inputs are distinguished from outputs. (This is, of course, necessarily the case in a programming language.) In that context, there are restrictions on the use of outputs. It is not possible, for instance, to have two outputs offered in an external choice, since in this case, we have a nondeterminism as to the choice of output communication that is going to be carried out. In abstract models, on the other hand, such nondeterminism is not a problem.

In [15], the lack of inputs and outputs in CSP is handled by defining a notion of test execution that takes this issue into account. The direction of the events is used to determine how to carry out the tests and determine a verdict. All this is formally defined, but soundness cannot be justified in the framework of CSP.

There are several lines of future work. Recent work has extended **ioco** to the case where there are distributed observations, leading to the **dioco** implementation relation [12]. Like **ioco**, the **dioco** implementation relation is only defined for input-enabled implementations. In addition, most of the work in this area has assumed that specifications are also input-enabled and the generalisations to the case where the specification need not be input-enabled are rather complex [12]. Observing refusal of inputs might help simplify treatment of an input not being enabled, but only in quiescent states; this could lead to simpler and more general implementation relations for distributed systems.

We have observed that input-output failures refinement does not imply conformance under **ioco** because **ioco** allows partial observation of refusals before the end of a trace. The $\mathcal{RT}$ model for CSP allows the observation of a sequence of events and refusal sets, and so it should be possible to adapt it to the case where we distinguish between inputs and outputs as well, and in this case produce a refinement relation strictly stronger than **ioco**.

The testing theory of *Circus* [17], an algebra that combines Z [24] and CSP, is similar to that of CSP. It is based on symbolic tests, and already takes advantage of the conventions of CSP to represent inputs and outputs more compactly. To leverage the results here to that context, though, we need an input-output failures model in the UTP [13], the semantic framework of *Circus*.

References

1. G. Barrett. Model checking in practice: The T9000 Virtual Channel Processor. *IEEE TSE*, 21(2):69 – 78, 1995.

2. I. B. Bourdonov, A. Kossatchev, and V. V. Kuliamin. Formal conformance testing of systems with refused inputs and forbidden actions. *ENTCS*, 164(4):83 – 96, 2006.
3. L. B. Briones and E. Brinksma. Testing real-time multi input-output systems. In *ICFEM*, volume 3785 of *LNCS*, pages 264 – 279. Springer, 2005.
4. A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in CSP. In *ICFEM*, volume 4789 of *LNCS*, pages 151 – 170. Springer, 2007.
5. A. L. C. Cavalcanti, M.-C. Gaudel, and R. M. Hierons. Conformance Relations for Distributed Testing based on CSP. In *ICTSS*, LNCS. Springer, 2011.
6. A. L. C. Cavalcanti and R. M. Hierons. Testing with inputs and outputs in CSP - Extended version. Technical report, 2012. Available at www-users.cs.york.ac.uk/ alcc/CH12.pdf.
7. 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.
8. C. Fischer. How to Combine Z with a Process Algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM*. Springer, 1998.
9. M.-C. Gaudel. Testing can be formal, too. In *TAPSOFT*, volume 915 of *LNCS*, pages 82 – 96. Springer, 1995.
10. A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(1):18 – 25, 2002.
11. L. Heerink and J. Tretmans. Refusal Testing for Classes of Transition Systems with Inputs and Outputs. In *FORTE/PSTV*, volume 107 of *IFIP Conference Proceedings*, pages 23 – 38. Chapman & Hall, 1997.
12. R. M. Hierons, M. G. Merayo, and M. Núñez. Implementation relations and test generation for systems with distributed interfaces. *Distributed Computing*, 25(1):35 – 62, 2012.
13. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
14. G. Jones. *Programming in* occam 2. Prentice-Hall, 1988.
15. T. Kahsai, M. Roggenbach, and B.-H. Schlingloff. Specification-based testing for refinement. In *SEFM*, pages 237 – 246. IEEE Computer Society, 2007.
16. B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *ICSE'98*, pages 95 – 104. IEEE Computer Society Press, 1998.
17. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3 – 32, 2009.
18. M. Roggenbach. CSP-CASL: a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42 – 71, 2006.
19. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
20. A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.
21. A. Sampaio, S. Nogueira, and A. Mota. Compositional verification of input-output conformance via CSP refinement checking. In *ICFEM*, volume 5885 of *LNCS*, pages 20 – 48. Springer, 2009.
22. S. Schneider and H. Treharne. Communicating B Machines. In *ZB*, volume 2272 of *LNCS*, pages 416 – 435, 2002.
23. J. Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *TACAS*, volume 1055 of *LNCS*, pages 127 – 146. Springer, 1996.
24. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.