The Best of Both Worlds: Model-Driven Engineering Meets Model-Based Testing

P. H. M. van Spaendonck[1]
Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands

Tim A. C. Willemse[1]
Department of Mathematics and Computer Science,
Eindhoven University of Technology, The Netherlands
ESI (TNO), Eindhoven, The Netherlands

Abstract

We study the connection between stable-failures refinement and the ioco conformance relation. Both behavioural relations underlie methodologies that have gained traction in industry: stable-failures refinement is used in several commercial Model-Driven Engineering tool suites, whereas the ioco conformance relation is used in Model-Based Testing tools. Refinement-based Model-Driven Engineering approaches promise to generate executable code from high-level models, thus guaranteeing that the code upholds specified behavioural contracts. Manual testing, however, is still required to gain confidence that the model-to-code transformation and the execution platform do not lead to unexpected contract violations. We identify conditions under which also this last step in the design methodology can be automated using the ioco conformance relation and the associated tools.

2012 ACM Subject Classification Theory of computation; Software and its engineering → Formal software verification

Keywords and phrases stable-failures refinement, Model-Driven Engineering, input output conformance, Input Output Labelled Transition Systems, internal choice

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.4

Funding *P. H. M. van Spaendonck*: This publication is part of the PVSR project (project number 17933) of the MasCot research programme which is financed by the Dutch Research Council (NWO).

Acknowledgements The authors would like to thank Rutger van Beusekom (Verum) for pointing them to the research question that led to this work. Furthermore, the authors would like to thank Jan Tretmans, and the anonymous referees, for feedback on an earlier version of this work.

1 Introduction

Formal Methods excel in eliminating subtle issues in complex software and system designs. Unfortunately, they are often perceived as complicated and inaccessible. For long, this sentiment has been a major reason for the slow industrial uptake of such methods. At the same time, Model-Driven Engineering (MDE), which promotes the use of Domain-Specific Languages (DSL) and code generation from models written in such languages, has managed to gain traction. MDE's success is in large part due to the close to perfect fit of a DSL and its application domain, which is in sharp contrast to the gap between generic Formal Methods and their domain of application.

[1] corresponding author

Currently, we are witnessing the adoption of *formal* MDE approaches, in which the DSL is coupled to a design methodology that advocates a stepwise, compositional approach based on behavioural contracts (sometimes referred to as *service* or *behavioural interface* specification) of components. Commercial approaches of this kind are, e.g., Verum's Dezyne methodology [21, 20] and Cocotec's Coco platform [6].

Underlying the stepwise approach is typically a notion of refinement; for instance, the Dezyne methodology essentially utilises CSP's *stable-failures refinement* [15, 10]. The central idea is that the code that is generated from a model refines its behavioural contract, provided that the model refines the same contract. This way, the code for entire constellations of – guaranteed seemlessly cooperating – components can be generated with little effort.

One step often overlooked, however, is the fact that the model that is being verified is not identical to the code that is executed: even if the code generator is flawless, the behaviour of the component still depends on the execution platform, its operating system, the compilers used, *etcetera*. As a result, testing is still required to gain confidence in the correct execution of the generated code.

In practice, testing is still a largely manual and time-consuming activity; at best scripting is used to automatically execute a number of manually crafted test cases. Model-Based Testing (MBT) is a formal approach to testing that aims to improve on that situation. Tretmans' conformance theory [17, 18] is one of the most widely used testing theories, which has even found commercial use. As a starting point, MBT approaches take a formal specification, describing the system-under-test, and automatically derive tests from that specification, thus saving time on manually constructing and executing test cases, and maintaining these as the specification (and implementation) evolve.

To enable reasoning about implementations, formal approaches to testing typically assume that there is some (otherwise unknown) model with specific characteristics that underlies the actual implementation. This is sometimes referred to as the *testing assumption*. For instance, Tretmans [17, 18] assumes that implementations behave as input enabled Labelled Transition Systems with inputs and outputs. Weiglhofer and Wotawa [26] observe that this class of models is not quite suited in asynchronous settings and advocate *internal choice* Labelled Transition Systems with inputs and outputs. Such transition systems accept inputs only in states that are stable and no longer able to produce outputs. Crucially, implementations that are obtained through the MDE approach often fall in this class: these generally employ a *run-to-completion* semantics that assumes a component is ready for input only when it has finished processing the previous input.

Combining formal MDE approaches and MBT approaches seems natural and beneficial, but in practice, the two do not appear to match. Indeed, it is part of folklore that Tretmans' conformance theory, viz. ioco, is impossible to reconcile with theories of refinement such as the stable-failures refinement: as we also show in this paper, there are implementations that formally refine their specifications, but that nevertheless do not pass tests derived from such specifications. *Vice versa*, implementations that pass all tests derived from a given specification do not necessarily refine that specification.

At the same time, there are specifications and implementations for which stable-failures refinement and ioco both (do not) hold, suggesting there may be some room for combining the MBT and MDE methodologies in practice. We address this issue in this paper. More specifically, we study conditions under which Tretmans' ioco conformance relation can be used to assess the quality of implementations under the assumption (or guarantee) that the implementation is a stable-failures refinement of its specification. Our contributions are threefold:

- We characterise experiments that can be deduced from a specification – so-called *stable-failures testable traces* – for which ioco is guaranteed not to reject implementations that are a stable-failures refinement of that specification;
- We show that, surprisingly, for the class of internal choice Labelled Transition Systems of Weiglhofer and Wotawa [24, 26, 12, 13], the set of stable-failures testable traces coincides with Tretmans' suspension traces, implying that off-the-shelf ioco-based tooling can be used to test these implementations;
- We validate our theory in practice through a proof-of-concept implementation. In particular, we assess whether industrial-grade executable code, obtained using Verum's Dezyne methodology:
  - passes all tests automatically derived from its behavioural contract, and
  - fails tests when subtle mutations are introduced in the code.

Related Work. Several authors have attempted to equip the CSP theory with a testing theory. Cavalcanti and Gaudel [3] instantiate Gaudel's testing theory [7] for the (divergence-free fragment of the) CSP language and compare failures refinement to the *conf* relation. Their work, unlike ours, does not fundamentally distinguish inputs and outputs, contrary to, e.g., ioco. Sound and complete test suites for CSP's refinement relation are studied in [14]. In [4], and also later in [5], CSP is equipped with the notion of input and output. The authors use this distinction, in contrast to our work, to modify the stable-failures refinement to define a *new* refinement relation that is stronger than ioco on input enabled CSP processes. In [25], the authors give a denotational characterisation of an ioco-inspired conformance relation, in the context of a CSP-like process algebra. They show that, when applied to processes representing the suspension automata underlying a given specification and implementation, their relation coincides with Tretmans' ioco. Related to these approaches, in [11], the authors introduce a conformance relation called *CSP input-output conformance* to test systems that are both input and output enabled. They exploit use case templates to generate test cases by means of counterexamples to stable failures refinement. Finally, in [1], the authors coin *input-output tock-CSP refinement* and study its correspondence to a timed variant of ioco, called tioco [16], showing that the latter is weaker than their refinement relation.

In the broader scope, there have been several studies looking at the ioco relation from the perspective of refinement theories and game theory. For instance, in [23], the authors observe that ioco is non-compositional – in contrast to a proper refinement relation – prompting the authors to weaken the ioco relation. Their relation coincides with ioco when specifications have no under-specified inputs (for a more detailed discussion, we refer to, e.g. [19]). In [9], the authors compare ioco to alternating trace containment, a refinement relation in the setting of game theory and formal verification. They omit internal transitions (also known as *silent* steps) from their model, but their treatment does cover quiescence. The connection between testing theory and game theory had been previously studied by Van den Bos and Stoelinga [22].

Paper outline. Our paper is organised as follows. In Section 2, we introduce stable-failures refinement and Tretmans' ioco theory. Then, in Section 3 we introduce stable-failures testable traces and study their role in testing implementations that refine their specifications. In Section 4, we identify conditions that allow for proving stable-failures refinement using ioco. Section 5 we describe our experiments with the theory we developed, and we draw conclusions and sketch future work in Section 6.

2 Preliminaries

The behaviour of a system is typically formalised using (variations of) labelled transition systems (LTSs). Actions, taken from a sufficiently large alphabet Act, represent the observables of a system. We presuppose a constant $\tau \notin \mathsf{Act}$ to represent an unobservable action; the set Act_τ denotes the set $\mathsf{Act} \cup \{\tau\}$.

▶ **Definition 1.** *A labelled transition system (LTS) over* Act *is a tuple* $\langle S, \hat{s}, \rightarrow\rangle$*, where* S *is a set of states,* $\hat{s} \in S$ *is the initial state and* $\rightarrow \subseteq S \times \mathsf{Act}_\tau \times S$ *is the transition relation. We denote the set of LTSs over* Act *by* $\mathcal{LTS}(\mathsf{Act})$*.*

We often refer to a given LTS $\langle S, \hat{s}, \rightarrow\rangle$ by its initial state $\hat{s}$. We write $s \xrightarrow{x} s'$ rather than $(s, x, s') \in \rightarrow$; moreover, we write $s \xrightarrow{x}$ when $s \xrightarrow{x} s'$ for some s', and $s \not\xrightarrow{x}$ when $s \xrightarrow{x}$ does not hold. The transition relation is lifted to a relation over $S \times \mathsf{Act}_\tau^* \times S$ in the usual manner, and we lift the notation introduced for $\rightarrow$ accordingly. We say that a word $w \in \mathsf{Act}_\tau^*$ is a *concrete trace* of an LTS $\hat{s}$ iff $\hat{s} \xrightarrow{w}$, and we say that a state s is *reachable* exactly when $\hat{s} \xrightarrow{w} s$ for some concrete trace w.

A further generalisation of $\rightarrow$ to a relation over words of observable actions $\Longrightarrow \subseteq S \times \mathsf{Act}^* \times S$ is obtained as the smallest relation satisfying the following rules:

$$\frac{}{s \overset{\epsilon}{\Longrightarrow} s} \qquad \frac{s \overset{w}{\Longrightarrow} s'' \quad s'' \xrightarrow{x} s' \quad x \neq \tau}{s \overset{w\,x}{\Longrightarrow} s'} \qquad \frac{s \overset{w}{\Longrightarrow} s'' \quad s'' \xrightarrow{\tau} s'}{s \overset{w}{\Longrightarrow} s'}$$

We adopt the notational conventions we introduced earlier for $\rightarrow$ also for $\Longrightarrow$. The set of *traces* of a states s is denoted $\mathsf{Traces}(s) = \{w \in \mathsf{Act}^* \mid s \overset{w}{\Longrightarrow}\}$. For a set of states S', we define $\mathsf{Traces}(S') = \bigcup_{s' \in S'} \mathsf{Traces}(s')$.

▶ **Definition 2.** *Let* $\langle S, \hat{s}, \rightarrow\rangle$ *be an LTS. For arbitrary state* $s \in S$ *and set of states* $S' \subseteq S$*, we define:*

1. $\mathsf{init}(s) = \{x \in \mathsf{Act}_\tau \mid s \xrightarrow{x}\}$ *and* $\mathsf{init}(S') = \bigcup_{s' \in S'} \mathsf{init}(s')$*;*
2. $\mathsf{Sinit}(s) = \{x \in \mathsf{Act} \mid s \overset{x}{\Longrightarrow}\}$ *and* $\mathsf{Sinit}(S') = \bigcup_{s' \in S'} \mathsf{Sinit}(s')$*;*
3. $\mathsf{stable}(s)$ *iff* $\tau \notin \mathsf{init}(s)$*, and* $\mathsf{stable}(S')$ *iff for all* $s' \in S$ *we have* $\mathsf{stable}(s')$*.*

We say that an LTS $\langle S, \hat{s}, \rightarrow\rangle$ is *convergent* when none of its states $s \in S$ are divergent, i.e., no state in S is the start of an infinite sequence of τ-steps.

A set of observable actions $X \subseteq \mathsf{Act}$ is a *refusal* for a state s exactly when $\mathsf{init}(s) \cap X = \emptyset$. Given a state s, we say that the pair (w, X) is a *failure* for state s when there is some s' such that $\mathsf{stable}(s')$, $s \overset{w}{\Longrightarrow} s'$ and $\mathsf{init}(s') \cap X = \emptyset$. The set of failures of a state s is denoted $\mathsf{Failures}(s)$, and defined formally as follows:

$$\mathsf{Failures}(s) = \{(w, X) \in \mathsf{Act}^* \times 2^{\mathsf{Act}} \mid \exists s' : s \overset{w}{\Longrightarrow} s' \wedge \mathsf{stable}(s') \wedge X \cap \mathsf{init}(s') = \emptyset\}$$

We next recall a classical notion of refinement underlying process algebras such as CSP, see, e.g. [15, 10].

▶ **Definition 3.** *Let* $\langle S, \hat{s}, \rightarrow\rangle$ *be an LTS. For states* $s, t \in S$*, we define* $s \sqsubseteq_F t$ *iff* $\mathsf{Traces}(t) \subseteq \mathsf{Traces}(s)$ *and* $\mathsf{Failures}(t) \subseteq \mathsf{Failures}(s)$*. We say* t *is a* stable-failures refinement *of* s *iff* $s \sqsubseteq_F t$*.*

When interacting with an actual implementation, the initiative to communicate is often not symmetric: the implementation can receive stimuli from its environment and produce events that are to be consumed by the environment. We therefore refine the LTS model to incorporate a distinction between *inputs* and *outputs*.

▶ **Definition 4.** *An input-output labelled transition system over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *is an LTS* $\langle S, \hat{s}, \rightarrow\rangle$ *over* Act *in which* Act *is partitioned into a set* Act_I *of inputs and a set* Act_U *of outputs. We denote the set of input-output labelled transition systems (IOLTS) over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *by* $\mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$.

As a notational convention we distinguish inputs from outputs by adding question- (?) and exclamation-mark (!) symbols, respectively, in our examples. We stress that these decorations are *not* part of action names. States are *quiescent* when they are stable and refuse to produce output. Quiescence, defined formally below, is a crucial element in many testing theories, needed to disqualify implementations that fail to produce output when not expected.

▶ **Definition 5.** *Let* $\langle S, \hat{s}, \rightarrow\rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*, and let* $s \in S$*. We say that* s *is* quiescent*, denoted* $\delta(s)$*, iff* $\mathsf{stable}(s)$ *and* $\mathsf{init}(s) \cap \mathsf{Act}_U = \emptyset$.

We say that an IOLTS $\langle S, \hat{s}, \rightarrow\rangle$ is an *internal choice IOLTS* iff inputs are only specified in quiescent states; i.e., exactly when for all $s \in S$ for which $\mathsf{init}(s) \cap \mathsf{Act}_I \neq \emptyset$, also $\delta(s)$ holds true. We denote the set of internal choice IOLTSs over $(\mathsf{Act}_I, \mathsf{Act}_U)$ by $\mathcal{IOLTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$.

Quiescence is typically treated as an output of the system, i.e., an observable of an implementation under test. Let $\delta \notin \mathsf{Act}$ be a special constant denoting the observation of quiescence, and let Act_δ denote the set $\mathsf{Act} \cup \{\delta\}$.

▶ **Definition 6.** *Let* $\langle S, \hat{s}, \rightarrow\rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*, and let* $s \in S$.

- *The* outputs *enabled in* s*, denoted* $\mathsf{out}(s)$*, is defined as* $\mathsf{out}(s) = \{\delta \mid \delta(s)\} \cup (\mathsf{Act}_U \cap \mathsf{init}(s))$;
- *The* inputs *enabled in* s*, denoted* $\mathsf{in}(s)$*, is defined as* $\mathsf{in}(s) = \mathsf{Act}_I \cap \mathsf{Sinit}(s)$.

For a set of states $S' \subseteq S$*, we define* $\mathsf{out}(S') = \bigcup_{s' \in S'} \mathsf{out}(s')$ *and* $\mathsf{in}(S') = \bigcap_{s' \in S'} \mathsf{in}(s')$.

The notion of a *suspension trace* incorporates the observation of quiescence also in our observations of the behaviour of an implementation over time.

▶ **Definition 7.** *Let* $\langle S, \hat{s}, \rightarrow\rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*, and let* $s \in S$*. We say that a sequence of events* $w \in \mathsf{Act}_\delta^*$ *is a* suspension trace *of* s *iff* $w \in \mathsf{Traces}(s_\Delta)$ *in the IOLTS* $\Delta(\hat{s})$ *over* $(\mathsf{Act}_I, \mathsf{Act}_U \cup \{\delta\})$*, where* $\Delta(\hat{s}) = \langle S_\Delta, \hat{s}_\Delta, \rightarrow_\Delta\rangle$ *is defined as follows:*

- $S_\Delta = \{s'_\Delta \mid s' \in S\}$;
- $\rightarrow_\Delta = \{(s'_\Delta, x, s''_\Delta) \mid s' \xrightarrow{x} s''\} \cup \{(s', \delta, s') \mid \delta(s')\}$.

The set of suspension traces of a state $s \in S$ *is denoted* $\mathsf{STraces}(s)$.

We generalise the relation $\rightarrow_\Delta$ to $\Longrightarrow_\Delta$ as before and we allow ourselves to write $s \overset{w}{\Longrightarrow}_\Delta s'$, for states s, s' of an IOLTS $\langle S, \hat{s}, \rightarrow\rangle$, when we in fact mean $s_\Delta \overset{w}{\Longrightarrow}_\Delta s'_\Delta$.

▶ **Definition 8.** *Let* $\langle S, \hat{s}, \rightarrow\rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*. For states* $s \in S$ *and suspension traces* $w \in \mathsf{STraces}(s)$*, we define* s after $w = \{s' \in S \mid s \overset{w}{\Longrightarrow}_\Delta s'\}$*. For sets of states* $S' \subseteq S$ *we define* S' after $w = \bigcup_{s' \in S'} s'$ after w.

Formal testing theories usually build upon the assumption that an implementation can be captured adequately in a submodel of IOLTSs. We recall two such submodels, viz., the *input output transition systems*, used in Tretmans' testing theory [17, 18] and the *internal choice input output transition systems*, introduced by Weiglhofer and Wotawa [24, 26].

Tretmans' input-output transition systems are IOLTSs with the additional assumption that inputs will always be accepted. That is, implementations are assumed to be *input enabled*.

▶ **Definition 9.** *Let* $\langle S, \hat{s}, \rightarrow, s_0\rangle$ *be an IOLTS over* $(\mathsf{Act}_I, \mathsf{Act}_U)$*. A state* $s \in S$ *is* input-enabled *iff* $\mathsf{Act}_I \subseteq \mathsf{Sinit}(s)$*. The IOLTS* $\hat{s}$ *is an* input output transition system *(IOTS) iff every state* $s \in S$ *is input-enabled. We denote the class of input output transition systems ranging over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *by* $\mathcal{IOTS}(\mathsf{Act}_I, \mathsf{Act}_U)$*.*

Weiglhofer and Wotawa's model of internal choice input output transition systems relax the requirement that implementations must be input-enabled at all times. Instead, they require that only quiescent states are input-enabled, and inputs are only accepted in quiescent states. Their model better fits with implementations that rely on some form of *run to completion*.

▶ **Definition 10** (Internal choice IOTS)**.** *An IOLTS* $\langle S, \hat{s}, \rightarrow\rangle$ *is an* internal choice input output transition system *over* $(\mathsf{Act}_I, \mathsf{Act}_U)$ *if for all states* $s \in S$*:*

1. *if* $\delta(s)$*, then* $\mathsf{Act}_I \subseteq \mathsf{init}(s)$
2. *if* $\mathsf{init}(s) \cap \mathsf{Act}_I \neq \emptyset$ *then* $\delta(s)$*.*

We denote the class of internal choice input output transition systems over $(\mathsf{Act}_I, \mathsf{Act}_U)$ *by* $\mathcal{IOTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$*.*

Testing is used to assess whether a given implementation conforms to its specification. Several conformance relations have been proposed in the literature, and one of the most prominent ones is *input output conformance* by Tretmans [17, 18]. This conformance relation formalises when an implementation, assumed to behave as an input output transition system, complies to a given specification. Following e.g. [9], we assume here that implementations can behave, more generally, as input output labelled transition systems.

▶ **Definition 11.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$ *be (a model of) an implementation and specification, respectively. We say that* imp input output conforms *to* spec*, denoted* $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$*, iff for all* $w \in \mathsf{STraces}(\mathsf{spec})$ *we have:*

1. $\mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$*,*
2. $\mathsf{in}(\mathsf{imp}\ \mathsf{after}\ w) \supseteq \mathsf{in}(\mathsf{spec}\ \mathsf{after}\ w)$*.*

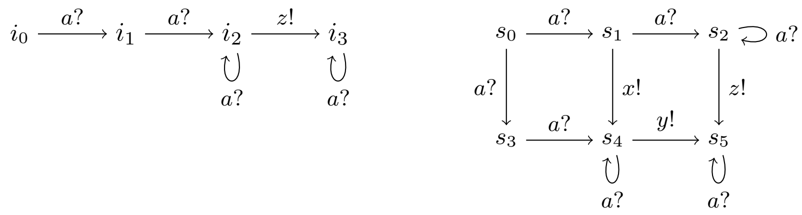
We remark that condition 2, on the inputs, can be dropped in the above definition in case the implementation is input enabled, thus simplifying to the definition that can be found in [17, 18]. After all, input enabledness guarantees that inputs can always be consumed by the implementation.

3 Testing Refinements of Specifications

Refinement relations are particularly useful in a design methodology in which a system is successively refined into smaller components, where, at each step, the relevant artefacts can be related by a stable-failures refinement. Once the models for (sub)components are sufficiently detailed and simple, implementing these as executable code should be reasonably straightforward and is even done automatically in formal MDE approaches.

Despite the simplicity and details of these models, the conversion to executable code may introduce bugs. Even if no bugs are introduced in this step, the platform on which the code runs may inject issues not foreseen at the time of the design. Conformance testing is therefore a step that cannot be omitted, but as the following example illustrates, the ioco-conformance relation may flag implementations to be incorrect, despite these being correct with respect to stable-failures refinement.

▶ **Example 12.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).



Observe that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ holds true. However, imp ioco spec does not hold, since $\mathsf{out}(s_0 \text{ after } a\,\delta\,a) = \{y\}$, whereas $\mathsf{out}(i_0 \text{ after } a\,\delta\,a) = \{z\}$. ⌟

Conceptually, the non-conformance in the above example is caused by the ability to continue testing beyond observations of quiescence. This suggests that, in general, we cannot safely test the full specification for all its suspension traces. The question thus arises what subset of the behaviour, modelled by a specification, is available to us for testing. We coin a set of suspension traces for which we subsequently argue that testing for these cannot lead to verdicts that conflict with previously established refinements.

▶ **Definition 13.** *Let* $\mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$ *be an arbitrary IOLTS. A suspension trace* w *of* spec *is* stable-failures testable *exactly when for all prefixes* $v\,\delta\,x$ *of* w*, with* $x \in \mathsf{Act}_\delta$*, we have* $\mathsf{spec} \text{ after } v\,x = \mathsf{spec} \text{ after } v\,\delta\,x$*. The set of all stable-failures testable suspension traces is denoted* $\mathsf{TTraces}(\mathsf{spec})$*.*

One may remark that the set Act_δ that x may range over in the above definition is too liberal. Indeed, since suspension traces are anomaly-free [27], x cannot be an output. Restricting the set of symbols that x ranges over to $\mathsf{Act}_I \cup \{\delta\}$ would therefore yield an equivalent, though in practice somewhat more cumbersome, definition.

We start by noting two relevant properties of the set of stable-failures testable traces of a specification.

▶ **Lemma 14.** *We have* $\mathsf{Traces}(\mathsf{spec}) \subseteq \mathsf{TTraces}(\mathsf{spec})$*.*

Proof. Pick some arbitrary $w \in \mathsf{Traces}(s)$. Since $\mathsf{Traces}(s) \subseteq \mathsf{STraces}(s)$, also $w \in \mathsf{Traces}(s)$. Moreover, since $w \in \mathsf{Traces}(s)$, also $w \in \mathsf{Act}^*$. Since there is no prefix of the shape $v\,\delta\,x$ in $w \in \mathsf{Act}^*$, we find that w is stable-failures testable. Hence $w \in \mathsf{TTraces}(\mathsf{spec})$. ◀

▶ **Lemma 15.** *The set* $\mathsf{TTraces}(\mathsf{spec})$ *is prefix closed.*

Proof. Pick some $w \in \mathsf{TTraces}(\mathsf{spec})$, and let w' be a prefix of w. Consider an arbitrary prefix $v\,\delta\,x$ of w'. Then $v\,\delta\,x$ is also a prefix of w. Since $w \in \mathsf{TTraces}(\mathsf{spec})$ we therefore have $\mathsf{spec} \text{ after } v\,x = \mathsf{spec} \text{ after } v\,\delta\,x$. But then also $w' \in \mathsf{TTraces}(\mathsf{spec})$. ◀

We next introduce the operators $\overline{\overline{w}}$ and $\overline{w}$ on suspension traces. In essence, these operators remove all δ-symbols (respectively, all but a terminal δ-symbol, if present) from a suspension trace.

▶ **Definition 16.** *Let* $x \in \mathsf{Act}_\delta$*,* $y \in \mathsf{Act}$*,* $v \in \mathsf{Act}^*$ *and* $w \in \mathsf{Act}_\delta^+$*. We define the operators* $\overline{\overline{\_}} : \mathsf{Act}_\delta^* \to \mathsf{Act}_\delta^*$ *and* $\overline{\_} : \mathsf{Act}_\delta^* \to \mathsf{Act}^*$ *as follows:*

$$\bar{\bar{\epsilon}} = \epsilon, \qquad \overline{\overline{y\,v}} = y\,\bar{\bar{v}}, \qquad \overline{\overline{\delta\,v}} = \bar{\bar{v}}$$

$$\bar{\epsilon} = \epsilon, \qquad \bar{x} = x, \qquad \overline{y\,w} = y\,\overline{w}, \qquad \overline{\delta\,w} = \overline{w}$$

Observe that in case $w \in \mathsf{Act}^*$, we have $\overline{w} = \overline{\overline{w}} = w$. In case $w \in \mathsf{Act}_\delta^* \mathsf{Act}^+$, we have $\overline{w} = \overline{\overline{w}}$, and in case $w \in \mathsf{Act}_\delta^* \delta^+$ we have $\overline{w} = \overline{\overline{w}}\,\delta$.

▶ **Lemma 17.** *Let* $\mathsf{spec} = \langle S, \hat{s}, \rightarrow \rangle$ *be an arbitrary IOLTS. For all* $w \in \mathsf{TTraces}(\mathsf{spec})$*,* $\mathsf{spec}\ \mathsf{after}\ w = \mathsf{spec}\ \mathsf{after}\ \overline{w}$*.*

Proof. The proof proceeds by means of an induction on the number of δ's appearing in w.

- Base case: w contains no δ-symbols. Then $w \in \mathsf{Traces}(\mathsf{spec})$ and since $\overline{w} = w$ for traces, we immediately find the desired $\mathsf{spec}\ \mathsf{after}\ w = \mathsf{spec}\ \mathsf{after}\ \overline{w}$.
- Induction: suppose that for all $z \in \mathsf{TTraces}(\mathsf{spec})$, containing n δ-symbols, we have $\mathsf{spec}\ \mathsf{after}\ z = \mathsf{spec}\ \mathsf{after}\ \overline{z}$. Pick some $w \in \mathsf{TTraces}(\mathsf{spec})$ containing $n+1$ δ-symbols. Then w must be of the shape $v\,\delta\,u$, with $u \in \mathsf{Act}^*$, and v containing n δ-symbols. We distinguish two cases:
  - Case $u = \epsilon$. Then $\mathsf{spec}\ \mathsf{after}\ w = \mathsf{spec}\ \mathsf{after}\ v\,\delta = (\mathsf{spec}\ \mathsf{after}\ v)\ \mathsf{after}\ \delta$ By induction, the latter is equal to $(\mathsf{spec}\ \mathsf{after}\ \overline{v})\ \mathsf{after}\ \delta$, which is equivalent to $\mathsf{spec}\ \mathsf{after}\ \overline{v}\,\delta$. We distinguish two further cases:
    * Case $\overline{v} \in \mathsf{Traces}(\mathsf{spec})$. Then $\overline{v}\,\delta = \overline{v\,\delta} = \overline{w}$, and consequently, $\mathsf{spec}\ \mathsf{after}\ \overline{v}\,\delta = \mathsf{spec}\ \mathsf{after}\ \overline{w}$.
    * Case $\overline{v} \notin \mathsf{Traces}(\mathsf{spec})$. Then $\overline{v} = v'\,\delta$ for some $v' \in \mathsf{Traces}(\mathsf{spec})$ and therefore $\overline{v}\,\delta = v'\,\delta\,\delta$. Observe that we have $\mathsf{spec}\ \mathsf{after}\ v'\,\delta\,\delta = \mathsf{spec}\ \mathsf{after}\ v'\,\delta = \mathsf{spec}\ \mathsf{after}\ \overline{v\,\delta} = \mathsf{spec}\ \mathsf{after}\ \overline{w}$.

    In both cases, we are done.
  - Case $u \neq \epsilon$. We necessarily have $u = x\,u'$ for some x and u'. Then, by Definition 13, we have $\mathsf{spec}\ \mathsf{after}\ w = \mathsf{spec}\ \mathsf{after}\ v\,\delta\,x\,u' = \mathsf{spec}\ \mathsf{after}\ v\,x\,u'$. Since $v\,x\,u'$ contains exactly n δ-symbols, we may conclude, by induction that $\mathsf{spec}\ \mathsf{after}\ v\,x\,u' = \mathsf{spec}\ \mathsf{after}\ \overline{v\,x\,u'}$. But $\overline{v\,x\,u'} = \overline{w}$, so we may conclude $\mathsf{spec}\ \mathsf{after}\ w = \mathsf{spec}\ \mathsf{after}\ \overline{w}$. ◀

▶ **Definition 18.** *We say that an IOLTS* spec *is stable-failures testable exactly when it satisfies* $\mathsf{STraces}(\mathsf{spec}) = \mathsf{TTraces}(\mathsf{spec})$*.*

It may be clear that not every IOLTS is stable-failures testable. For instance, the specification depicted in Example 12 contains suspension traces that are not stable-failures testable: the sequence $a\,\delta\,a$, which we used to illustrate the non-conformance of the implementation to the specification is not stable-failures testable, since $s_0\ \mathsf{after}\ a\,\delta\,a = \{s_4\} \neq \{s_2, s_4\} = s_0\ \mathsf{after}\ a\,a$. On the other hand, the implementation depicted in the same example is stable-failures testable. The class of internal choice IOLTSs also turns out to be stable-failures testable, as asserted by the theorem below.

▶ **Theorem 19.** *Every internal choice IOLTS is stable-failures testable.*

Proof. Clearly, $\mathsf{TTraces}(\mathsf{spec}) \subseteq \mathsf{STraces}(\mathsf{spec})$, so it suffices to prove $\mathsf{STraces}(\mathsf{spec}) \subseteq \mathsf{TTraces}(\mathsf{spec})$. This can be shown using an induction on the length of the suspension traces.

- Base case $w = \epsilon$. Since $\epsilon \in \mathsf{Traces}(\mathsf{spec}) \subseteq \mathsf{TTraces}(\mathsf{spec})$, we are done.
- Suppose that for $w \in \mathsf{STraces}(\mathsf{spec})$ of length n, we have $w \in \mathsf{TTraces}(\mathsf{spec})$. Let $x \in \mathsf{Act}_\delta$ be such that $w\,x \in \mathsf{STraces}(\mathsf{spec})$. Let $v\,\delta\,y$ be a prefix of $w\,x$. If $v\,\delta\,y$ is a prefix of w, then we may conclude $\mathsf{spec}\ \mathsf{after}\ v\,y = \mathsf{spec}\ \mathsf{after}\ v\,\delta\,y$ from our induction hypothesis and we are done.

  So suppose that $v\,\delta\,y = w\,x$. It now suffices to prove that $\mathsf{spec}\ \mathsf{after}\ v\,y = \mathsf{spec}\ \mathsf{after}\ v\,\delta\,y$. Note that $\mathsf{spec}\ \mathsf{after}\ v\,y \supseteq \mathsf{spec}\ \mathsf{after}\ v\,\delta\,y$ follows from the fact that observations of δ do not change state, so it suffices to prove $\mathsf{spec}\ \mathsf{after}\ v\,y \subseteq \mathsf{spec}\ \mathsf{after}\ v\,\delta\,y$. Pick some $s \in \mathsf{spec}\ \mathsf{after}\ v\,y$. From $w\,x = v\,\delta\,y \in \mathsf{STraces}(\mathsf{spec})$ we may conclude that $y \notin \mathsf{Act}_U$. We distinguish two cases:

- Case $y = \delta$. Then it immediately follows that also $s \in \mathsf{spec\ after}\ v\,\delta\,y$ and we are done.
- Case $y \neq \delta$. This implies that $y \in \mathsf{Act}_I$. Since spec is an internal choice IOLTS, we find that there must be some $s' \in \mathsf{spec\ after}\ v$ such that $\delta(s')$ and $s' \xRightarrow{y}_\Delta s$. Let s' be such. Since $s' \xRightarrow{\delta}_\Delta s'$, we may conclude that also $s \in \mathsf{spec\ after}\ v\,\delta\,y$. ◀

We next formally relate the failures refinement theory to the input output conformance testing theory. Lemma 20 states that the outputs of implementations that are a stable-failures refinement of a given specification can be safely tested using stable-failures testable suspension traces. Likewise, Lemma 21, states that the inputs of convergent implementations that are a stable-failures refinement of a given specification can be safely tested using stable-failures testable suspension traces.

▶ **Lemma 20.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$. *Assume that* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *holds true. Then* $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$ *for all* $w \in \mathsf{TTraces}(\mathsf{spec})$.

Proof. Suppose that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. Towards a contradiction, assume that for some $w \in \mathsf{TTraces}(\mathsf{spec})$ we do not have $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$. Without loss of generality, assume that w is the shortest such trace. This implies, in particular, that w is not of the form $v\,\delta$, since such a suspension trace cannot give rise to the desired contradiction, and therefore $\overline{w} \in \mathsf{Traces}(\mathsf{spec})$. Note that we also can conclude that $\mathsf{out}(\mathsf{imp\ after}\ w) \neq \emptyset$ and hence $w \in \mathsf{STraces}(\mathsf{imp})$. Since imp is quiescence-reducible [27], we therefore also have $\overline{w} \in \mathsf{Traces}(\mathsf{imp})$. By definition, $\mathsf{imp\ after}\ w \subseteq \mathsf{imp\ after}\ \overline{w}$. Consequently, $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{imp\ after}\ \overline{w})$. Furthermore, using Lemma 17 we may conclude that $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$, so also $\mathsf{out}(\mathsf{spec\ after}\ w) = \mathsf{out}(\mathsf{spec\ after}\ \overline{w})$.

Let $X = \mathsf{out}(\mathsf{imp\ after}\ \overline{w}) \setminus \mathsf{out}(\mathsf{spec\ after}\ \overline{w})$. We distinguish two cases:

- Case $\delta \in X$. Then, $(\overline{w}, \mathsf{Act}_U) \in \mathsf{Failures}(\mathsf{imp})$, but $(\overline{w}, \mathsf{Act}_U) \notin \mathsf{Failures}(\mathsf{spec})$. Since $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$, this cannot be the case. Contradiction.
- Case $\delta \notin X$. Pick $x \in X$. Then $\overline{w}\,x \in \mathsf{Traces}(\mathsf{imp})$, but $\overline{w}\,x \notin \mathsf{Traces}(\mathsf{spec})$. Again, since $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$, this cannot be the case. Contradiction.

Since both cases lead to a contradiction, we may conclude that for all $w \in \mathsf{TTraces}(\mathsf{spec})$ we have $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$. ◀

▶ **Lemma 21.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$. *Assume* imp *is convergent and assume* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *holds true. Then* $\mathsf{in}(\mathsf{spec\ after}\ w) \subseteq \mathsf{in}(\mathsf{imp\ after}\ w)$ *for all* $w \in \mathsf{TTraces}(\mathsf{spec})$.

Proof. Assume that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. Suppose that for $w \in \mathsf{TTraces}(\mathsf{spec})$, $\mathsf{in}(\mathsf{spec\ after}\ w) \subseteq \mathsf{in}(\mathsf{imp\ after}\ w)$ does not hold. Note that this implies that $\mathsf{in}(\mathsf{spec\ after}\ w) \neq \emptyset$. Pick such w and some input $a \in \mathsf{in}(\mathsf{spec\ after}\ w) \setminus \mathsf{in}(\mathsf{imp\ after}\ w)$. By definition, this means that for all $s \in \mathsf{spec\ after}\ w$ we have $s \xRightarrow{a}$. By Lemma 17, $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{w}$, so also $s \xRightarrow{a}$ for all $s \in \mathsf{spec\ after}\ \overline{w}$. Observe that this also implies that for all stable states $t \in \mathsf{spec\ after}\ \overline{w}$, if any, we have $t \xrightarrow{a}$. We distinguish two cases:

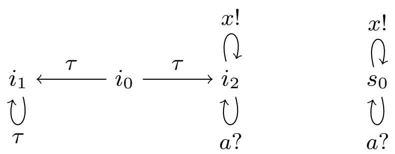
- $\overline{w} \notin \mathsf{Traces}(\mathsf{spec})$. Then $\overline{w} = \overline{\overline{w}}\,\delta$ and since $\mathsf{spec\ after}\ w = \mathsf{spec\ after}\ \overline{\overline{w}}\,\delta \neq \emptyset$, there is some $t \in \mathsf{spec\ after}\ \overline{\overline{w}}\,\delta$ satisfying $\delta(t)$, and which is therefore stable. Since for every stable state $t \in \mathsf{spec\ after}\ \overline{\overline{w}}\,\delta$ we have $t \in \mathsf{spec\ after}\ \overline{\overline{w}}$, we may conclude that $(\overline{\overline{w}}, \{a\}) \notin \mathsf{Failures}(\mathsf{spec})$.
- $\overline{w} \in \mathsf{Traces}(\mathsf{spec})$. Since in that case $\overline{w} = \overline{\overline{w}}$, we again conclude that $(\overline{\overline{w}}, \{a\}) \notin \mathsf{Failures}(\mathsf{spec})$.

From the above, we thus conclude that $(\overline{\overline{w}}, \{a\}) \notin \mathsf{Failures}(\mathsf{spec})$. We will next argue that $(\overline{\overline{w}}, \{a\}) \in \mathsf{Failures}(\mathsf{imp})$. Since this contradicts $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$, we may conclude that $\mathsf{in}(\mathsf{spec\ after}\ w) \subseteq \mathsf{in}(\mathsf{imp\ after}\ w)$, finishing the proof.

Concerning the remaining proof obligation $(\overline{\overline{w}}, \{a\}) \in \mathsf{Failures}(\mathsf{imp})$, we reason as follows. Since $a \notin \mathsf{in}(\mathsf{imp\ after}\ w)$ and imp is convergent, we conclude that there must be some state $s \in \mathsf{imp\ after}\ w$ such that $\mathsf{stable}(s)$ and $s \overset{a}{\not\rightarrow}$. Let s be such a state. By definition, we have $\mathsf{imp\ after}\ w \subseteq \mathsf{imp\ after}\ \overline{\overline{w}}$, so also $s \in \mathsf{imp\ after}\ \overline{\overline{w}}$. But then $(\overline{\overline{w}}, \{a\}) \in \mathsf{Failures}(\mathsf{imp})$. ◀

One might wonder whether the convergence condition is strictly needed. The example below illustrates that this condition can indeed not be dropped in general.

▶ **Example 22.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).



Observe that $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ holds true. Moreover, note that due to the τ-loop, imp is not convergent. By Lemma 20, we find that for every $w \in \mathsf{STraces}(\mathsf{spec})$, we have $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$; this is readily checked. However, we have $\mathsf{in}(\mathsf{spec\ after}\ \epsilon) = \{a\} \neq \emptyset = \mathsf{in}(\mathsf{imp\ after}\ \epsilon)$. Consequently, imp ioco spec does not hold true. ⌟

The theorem below follows immediately from the two lemmata above.

▶ **Theorem 23.** *Let* $\mathsf{imp}, \mathsf{spec} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$. *Assume* imp *is convergent. If* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *then also for all* $w \in \mathsf{TTraces}(\mathsf{spec})$, *we have:*
1. $\mathsf{out}(\mathsf{imp\ after}\ w) \subseteq \mathsf{out}(\mathsf{spec\ after}\ w)$, *and*
2. $\mathsf{in}(\mathsf{imp\ after}\ w) \supseteq \mathsf{in}(\mathsf{spec\ after}\ w)$.

Theorem 23 specialises to standard ioco in case the specification is an internal choice IOLTS and the implementation is convergent, as claimed by the corollary below.

▶ **Corollary 24.** *Let* $\mathsf{spec} \in \mathcal{IOLTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$ *and* $\mathsf{imp} \in \mathcal{IOLTS}(\mathsf{Act}_I, \mathsf{Act}_U)$. *Suppose* imp *is convergent. Then* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *implies* imp ioco spec.

We finish with the observation that in case the specification is an internal choice IOLTS and the implementation is an internal choice IOTS, the requirement on the implementation being convergent can be dropped, see the corollary below.

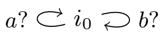
▶ **Corollary 25.** *Let* $\mathsf{spec} \in \mathcal{IOLTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$ *and* $\mathsf{imp} \in \mathcal{IOTS}^{\sqcap}(\mathsf{Act}_I, \mathsf{Act}_U)$. *Then* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ *implies* imp ioco spec.

4 Stable Failures Refinement through Testing

We next identify conditions under which we may conclude that a model of an implementation is a stable-failures refinement of a given specification after exhaustively testing a faithful implementation of that model.

Let us first observe that if the specification that is used for testing is not input enabled, we will not be able to establish a stable-failures refinement relation between the specification and the implementation. Since the ioco-conformance relation allows for partial specifications, only those parts that are specified are tested for, and other parts are ignored, resulting in potentially labelling such an implementation as one that conforms to its specification. As a result, inputs that are not specified cannot be excluded to be part of some conforming implementation and will thus lead to trace inclusion violations. This is illustrated by the following (trivial) example.

▶ **Example 26.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).



Clearly, we have imp ioco spec, but the trace $b \in \mathsf{Traces}(i_0)$ is not present in $\mathsf{Traces}(s_0)$, thus contradicting spec $\sqsubseteq_F$ imp. ⌟

Consequently we can only assess that an implementation refines a specification if the latter is "at least as input enabled" as the implementation that we are (black box) testing for. In Tretmans original testing theory, but also in Weiglhofer and Wotawa's theory, the input enabledness of the implementation is typically part of the testing assumption, which, depending on the applications at hand, state that the implementation is either always input enabled (IOTSs), or input enabled exactly (and only) in quiescent states (internal choice IOTSs). We therefore confine our analysis to implementations that can be modelled as an IOTS or an internal choice IOTS, and we study specifications that – in terms of their input enabledness – fit these assumptions. For these systems, we have the following observation:

▶ **Lemma 27.** *Let* spec, imp *be IOLTSs. Suppose that either:*
- *both* spec *and* imp *are IOTSs, or*
- *both* spec *and* imp *are internal choice IOTSs.*

Then imp ioco spec *implies* $\mathsf{Traces}(\mathsf{imp}) \subseteq \mathsf{Traces}(\mathsf{spec})$.

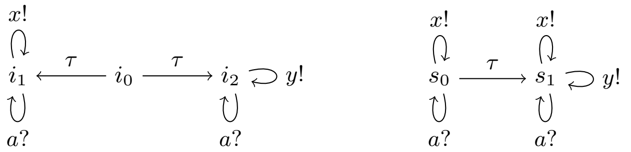
Proof. Suppose that imp ioco spec. Let $w \in \mathsf{Traces}(\mathsf{imp})$ be such that $w \notin \mathsf{Traces}(\mathsf{spec})$, and, without loss of generality, assume that there is no shorter trace. Observe that $w \neq \epsilon$, since ϵ is a weak trace of both imp and spec. Hence, w must be of the shape $v\,x$, for some trace $v \in \mathsf{Traces}(\mathsf{imp}) \cap \mathsf{Traces}(\mathsf{spec})$ and action $x \in \mathsf{Act}$. Let v and x be such.

We first argue that $x \notin \mathsf{Act}_I$. Observe that this follows trivially in case imp and spec are both IOTSs, since spec would be required to accept input a at any moment. In case spec is an internal choice IOTS, we reason as follows. Towards a contradiction, assume that $x \in \mathsf{Act}_I$. Then $v\,x \notin \mathsf{Traces}(\mathsf{spec})$ can only be the case when $\delta \notin \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ v)$, since spec is input enabled only (and exactly) in quiescent states. Since $v\,x \in \mathsf{Traces}(\mathsf{imp})$, we must conclude that $\delta \in \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ v)$. But this violates our assumption that imp ioco spec. Hence, also in case imp and spec are internal choice IOTSs, we have $x \notin \mathsf{Act}_I$.

Consequently, $x \in \mathsf{Act}_U$ and therefore $x \in \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ v)$. Since $v \in \mathsf{STraces}(\mathsf{spec})$ and imp ioco spec, we also find $x \in \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ v)$. This implies that $v\,x \in \mathsf{Traces}(\mathsf{spec})$. Contradiction. Hence, $\mathsf{Traces}(\mathsf{imp}) \subseteq \mathsf{Traces}(\mathsf{spec})$. ◀

In view of the above result, assuming some form of input enabledness of the specification is essential for guaranteeing trace inclusion, which is an essential part of the refinement relation. However, input enabledness does little to establish the other essential part of the refinement relation, viz., the inclusion of the set of failures. This has to do with the fact that refinement allows for observing the refusals of individual actions, contrary to the ioco conformance relation. The next example illustrates the issue. We remark that the example uses an implementation that behaves as an IOTS, but this can be modified easily to show the same issue in internal choice IOTSs.

▶ **Example 28.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).



Note that imp ioco spec; in particular, $\mathsf{out}(i_0 \text{ after } \epsilon) = \mathsf{out}(s_0 \text{ after } \epsilon)$. Clearly, $(\epsilon, \{y\}) \notin$ $\mathsf{Failures}(s_0)$ since stable state s_1 does not refuse y; of course, state s_0 does not offer action y, but since s_0 is unstable, its refusals are not taken into account. However, since i_1 is stable, $(\epsilon, \{y\}) \in \mathsf{Failures}(i_0)$. Therefore, spec $\sqsubseteq_F$ imp does not hold true. ⌟

The above example illustrates that, from the point of view of stable-failures refinement, output actions should be preserved and ultimately *determined*: τ-paths should eventually lead to states in which only "trivial output choices" can be made.

▶ **Definition 29.** *Let $\langle S, \hat{s}, \rightarrow\rangle$ be an IOLTS. We say that $\hat{s}$ is* ultimately determined *iff for all states $s \in S$ and all $x \in \mathsf{out}(s \text{ after } \epsilon)$ there is some $t \in s$ after ϵ such that $\mathsf{out}(t) = \{x\}$.*

Observe that the specification of Example 28 is not ultimately determined, since, e.g., there is no state $s \in s_0$ after ϵ such that $\mathsf{out}(s) = \{y\}$.

▶ **Proposition 30.** *For any IOTS* imp *and convergent, ultimately determined IOTS* spec *satisfying* imp ioco spec *we have* spec $\sqsubseteq_F$ imp.

Proof. Suppose that imp ioco spec holds true for IOTSs imp and spec, and that spec is both convergent and ultimately determined. We show that spec $\sqsubseteq_F$ imp; by Lemma 27, it suffices to prove that $\mathsf{Failures}(\mathsf{imp}) \subseteq \mathsf{Failures}(\mathsf{spec})$.

Towards a contradiction, assume that $\mathsf{Failures}(\mathsf{imp}) \not\subseteq \mathsf{Failures}(\mathsf{spec})$. Pick a failure $(w, X) \in \mathsf{Failures}(\mathsf{imp})$ such that $(w, X) \notin \mathsf{Failures}(\mathsf{spec})$. Observe that since $\mathsf{Traces}(\mathsf{imp}) \subseteq$ $\mathsf{Traces}(\mathsf{spec})$, $w \in \mathsf{Traces}(\mathsf{imp}) \cap \mathsf{Traces}(\mathsf{spec})$. Without loss of generality, assume that X is as large as possible: there is no Y such that $(w, Y) \in \mathsf{Failures}(\mathsf{imp}) \setminus \mathsf{Failures}(\mathsf{spec})$ such that $X \subset Y$. Then $\mathsf{imp} \overset{w}{\Longrightarrow} t$ such that $\mathsf{stable}(t)$ holds true and $\mathsf{init}(t) \cap X = \emptyset$.

Note that since imp is an IOTS and t is stable, we have $\mathsf{Act}_I \subseteq \mathsf{init}(t)$ so $X \subseteq \mathsf{Act}_U$. Because imp ioco spec, we have $\mathsf{out}(t) \subseteq \mathsf{out}(\mathsf{imp} \text{ after } w) \subseteq \mathsf{out}(\mathsf{spec} \text{ after } w)$. So there must be a state $s \in \mathsf{spec}$ after w such that $\mathsf{out}(t) \cap \mathsf{out}(s) \neq \emptyset$. Let s be such a state, and pick some $x \in \mathsf{out}(t) \cap \mathsf{out}(s)$. Since spec is convergent, all τ-paths are finite and end in a stable state. Because spec is ultimately determined there must be some stable state $s' \in s$ after ϵ such that $\mathsf{out}(s') = \{x\}$. Then $\mathsf{out}(s') \subseteq \mathsf{out}(t)$, and consequently, $\mathsf{init}(s') \cap X = \emptyset$. But then also $(w, X) \in \mathsf{Failures}(\mathsf{spec})$. Contradiction, so $\mathsf{Failures}(\mathsf{imp}) \subseteq \mathsf{Failures}(\mathsf{spec})$ and therefore spec $\sqsubseteq_F$ imp. ◀

Note that there is a rather straightforward reason why we cannot simply drop the assumption on the specification being convergent; see the example below.

▶ **Example 31.** Consider the implementation imp, with initial state i_0, depicted below (left) and the specification spec, with initial state s_0, depicted below (right).

Observe that $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$. Moreover, spec is trivially ultimately determined: $s_0\ \mathsf{after}\ \epsilon = \{s_0\}$ and the only output action enabled in s_0 is x. Because s_0 is not stable, we have $(\epsilon, \{y\}) \notin \mathsf{Failures}(s_0)$. On the other hand, $(\epsilon, \{y\}) \in \mathsf{Failures}(i_0)$, so we cannot have $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. ⌟

We finish this section with a similar statement for the internal choice testing theory.

▶ **Proposition 32.** *For any internal choice IOTS* imp *and convergent, ultimately determined internal choice IOTS* spec *satisfying* $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$ *we have* $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$.

Proof. Let imp be an internal choice IOTS and spec a convergent, determined internal choice IOTS. Assume that $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$ holds true. We argue that also $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$ holds true. Towards a contradiction, suppose that $\mathsf{spec} \not\sqsubseteq_F \mathsf{imp}$. Then, by Lemma 27, $\mathsf{Failures}(\mathsf{imp}) \not\subseteq \mathsf{Failures}(\mathsf{spec})$.

Suppose $\mathsf{Failures}(\mathsf{imp}) \not\subseteq \mathsf{Failures}(\mathsf{spec})$. Pick a failure $(w, X) \in \mathsf{Failures}(\mathsf{imp})$ such that $(w, X) \notin \mathsf{Failures}(\mathsf{spec})$. Then $\mathsf{imp} \overset{w}{\Longrightarrow} t$ such that $\mathsf{stable}(t)$ holds true and $\mathsf{init}(t) \cap X = \emptyset$. Note that since imp is an internal choice IOTS and $\mathsf{stable}(t)$ holds true, we have either $\mathsf{init}(t) = \mathsf{Act}_I$ or $\emptyset \subset \mathsf{init}(t) \subseteq \mathsf{Act}_U$.

- Suppose that $\mathsf{init}(t) = \mathsf{Act}_I$. Because imp is an internal choice IOTS, $\delta \in \mathsf{out}(t)$ and therefore $\delta \in \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w)$. Since $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$, also $\delta \in \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$ and hence $w\,\delta \in \mathsf{STraces}(\mathsf{spec})$. This means that there must be some state s such that $\mathsf{spec} \overset{w}{\Longrightarrow} s$, $\mathsf{stable}(s)$ and $\mathsf{init}(s) \cap \mathsf{Act}_U = \emptyset$. Pick such a state s. Since spec is an internal choice IOTS, $\mathsf{init}(s) = \mathsf{Act}_I$. Note that also $\mathsf{init}(t) = \mathsf{Act}_I$ and therefore $\mathsf{init}(s) = \mathsf{init}(t)$. But then also $\mathsf{init}(s) \cap X = \emptyset$. Consequently, $(w, X) \in \mathsf{Failures}(\mathsf{spec})$. Contradiction.
- Suppose that $\emptyset \subset \mathsf{init}(t) \subseteq \mathsf{Act}_U$. Then $\mathsf{Act}_I \subseteq X$. Moreover, because $\mathsf{imp}\ \mathsf{ioco}\ \mathsf{spec}$, we have $\emptyset \subset \mathsf{init}(t) \subseteq \mathsf{out}(\mathsf{imp}\ \mathsf{after}\ w) \subseteq \mathsf{out}(\mathsf{spec}\ \mathsf{after}\ w)$. So, there must be some state s such that $\mathsf{spec} \overset{w}{\Longrightarrow} s$ and $\mathsf{init}(t) \cap \mathsf{out}(s) \neq \emptyset$. Let s be such a state. Since spec is ultimately determined, we find that for all $x \in \mathsf{out}(s)$, there must be some $s' \in s\ \mathsf{after}\ \epsilon$ such that $\mathsf{out}(s') = \{x\}$. Pick some $x \in \mathsf{init}(t) \cap \mathsf{out}(s)$, and let s' be such that $s' \in s\ \mathsf{after}\ \epsilon$ and $\mathsf{out}(s') = \{x\}$. This means that $\mathsf{out}(s') \subseteq \mathsf{init}(t)$. Since spec is convergent and ultimately determined, we may assume that s' is stable. Observe that s' cannot be quiescent since $\mathsf{out}(s') \subseteq \mathsf{init}(t) \subseteq \mathsf{Act}_U$. Since $\mathsf{spec} \in \mathcal{IOTS}^{\sqcap}$, we therefore find that $\mathsf{Act}_I \cap \mathsf{init}(s') = \emptyset$, and hence $\mathsf{init}(s') \subseteq \mathsf{init}(t)$. Consequently, $\mathsf{init}(s') \cap X \subseteq \mathsf{init}(t) \cap X = \emptyset$. From this, we can conclude that $(w, X) \in \mathsf{Failures}(\mathsf{spec})$. Contradiction.

Hence, $\mathsf{Failures}(\mathsf{imp}) \subseteq \mathsf{Failures}(\mathsf{spec})$, and therefore $\mathsf{spec} \sqsubseteq_F \mathsf{imp}$. ◀

5 A Small Experiment: Testing Dezyne using mCRL2

As a practical validation of our theory, we apply MBT to a specification and implementation stemming from an industrial model of a multi-component controller at Philips Image Guided Therapy systems. The implementation has been generated from specifications in the Dezyne formal modelling DSL [21, 21]. In the Dezyne development methodology, a system is described as a hierarchical composition of components by specifying:

- a set of behavioural contracts, called *interfaces*. Each interface provides an abstraction of a component, the so-called *provided interface* of the component, and
- a behavioural model (a state machine) that describes how a component realises its behavioural contract, by interacting with subcomponents. The ports via which the component connects to subcomponents are called *required ports*, and by association, the behavioural contracts upon which the component relies are therefore referred to as *required interfaces*.

Table 1 test run results of MBT applied to correct and faulty code-generated implementation.

average	correct impl.	Mutation 1	2	3	4	5	6
detection rate	0%	96%	100%	100%	100%	100%	100%
actions required	200	45	41	9	8	10	17
state coverage	96%	80%	85%	46%	46%	63%	53%

The formal check that takes place in Dezyne, before generating code, is whether a component complies to its provided interface. This check is answered by verifying whether the IOLTS induced by the provided interface is stable-failures refined by the IOLTS obtained by combining the IOLTS underlying the component and the IOLTSs underlying the behavioural contracts of the subcomponents. The actual stable-failures refinement check is conducted using the mCRL2 toolset [2, 8]. In case a component is found to comply to its provided interface (and only then), the behavioural model of the component is fully automatically converted into an equivalent executable C++ program. This way, a correct-by-construction system can be built from the ground-up, or top-down by specifying, in a step-wise manner, desired provided interfaces and introducing (sub)components that "implement" these.

For the system that we study in this section, we do not have access to the implementation of the subcomponents for the required interfaces of our component, but we do have access to their behavioural contracts and the code that was generated from the main component itself. In our experiments, we therefore mimic the behaviour of the subcomponents via a simulator that utilises the IOLTSs of the behavioural contracts of the subcomponents instead. This yields so-called smart stubs. The specification IOLTS of the multi-component controller consists of 25 unique states and 54 unique transitions and is stable-failures testable. As per our theory, the MBT algorithm should not find any non-conformance since the implementation (the component together with the smart stubs) is a stable-failures refinement of the specification (the provided interface). Hence, if a non-conformance is found, the implementation does not reflect the model of the component that was proved to comply to its behavioural contract, and the non-conformance thus signals an actual issue with the executable or the platform.

We are interested in assessing whether we can detect erroneous implementations of the specification using ioco-based MBT techniques. To this end, we test the correct implementation and, in addition, 6 manually created, faulty mutants thereof. The first five faulty mutants are obtained by altering the implementation of the component such that a single randomly chosen input which would normally result in a state change, now performs no actual code execution, and thus results in no state change in the implementation. For the sixth mutant, each provided interface has been given a preset (1/10) chance of remaining idle, instead of providing a response when triggered, which should result in a non-conforming quiescence observation.

Using an on-the-fly MBT algorithm, which implements the original ioco test algorithm [17, 18] in mCRL2, we generated and executed 100 test runs, each consisting of up-to 200 observable actions (including quiescence) for each mutant and for the correct implementation. The results of this experiment are shown in Table 1. For each set of 100 test runs, we measured the percentage of runs that detected a non-conformance, the average number of observable actions (including quiescence) required to observe that non-conformance or terminate (in the case that no non-conformance is detected) and the average specification state coverage, i.e., unique states visited during a test-run. We observe that no non-conformances were detected when testing the correct implementation. In virtually all of the test runs on incorrect implementations a non-conformance was detected when using incorrect implementations, once more confirming the practical relevance of automated testing.

6 Conclusions

We studied the stable-failures refinement relation [15] and its relation to the ioco conformance testing relation by Tretmans [17, 18]. In particular, we identified a set of experiments – called *stable-failures testable traces* – derivable from a specification, for which ioco does not falsely flag implementations as incorrect when these implementations have been shown to refine the specification, thus addressing a major obstacle in applying Model-Based Testing techniques in the Model-Driven Engineering development method. Furthermore, we showed that for internal choice input output transition systems, these experiments coincide with the full set of experiments usually associated with the ioco testing theory. To better understand the limitations of ioco-based testing, we additionally identify conditions under which exhaustive testing can establish that the implementation refines the specification used for testing.

We did not explore how to implement our testing theory efficiently for specifications whose stable-failures testable traces are a proper subset of the suspension traces; this is left for future work. For finite specifications, deriving stable-failures testable traces is easily achieved by means of a determinisation-like algorithm, constructing a *Suspension Automaton* [17, 27] and exploring that structure. For infinite specifications, efficiently deriving and selecting such stable-failures testable traces *on-the-fly* would allow to combine the testing methodology with other *on-the-fly* testing algorithms.

References

1 James Baxter, Ana Cavalcanti, Maciej Gazda, and Robert M. Hierons. Testing using CSP models: Time, inputs, and outputs. *ACM Trans. Comput. Log.*, 24(2):17:1–17:40, 2023. doi:10.1145/3572837.

2 Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs, and Tim A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In *TACAS (2)*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019. doi:10.1007/978-3-030-17465-1_2.

3 Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in CSP. In *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 2007. doi:10.1007/978-3-540-76650-6_10.

4 Ana Cavalcanti and Robert M. Hierons. Testing with inputs and outputs in CSP. In *FASE*, volume 7793 of *Lecture Notes in Computer Science*, pages 359–374. Springer, 2013. doi:10.1007/978-3-642-37057-1_26.

5 Ana Cavalcanti, Robert M. Hierons, and Sidney C. Nogueira. Inputs and outputs in CSP: A model and a testing theory. *ACM Trans. Comput. Log.*, 21(3):24:1–24:53, 2020. doi:10.1145/3379508.

6 Cocotec. Coco platform. https://cocotec.io/, 2023. Accessed: 01 May 2023.

7 Marie-Claude Gaudel. Testing can be formal, too. In *TAPSOFT*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995. doi:10.1007/3-540-59293-8_188.

8 Jan Friso Groote, Jeroen J. A. Keiren, Bas Luttik, Erik P. de Vink, and Tim A. C. Willemse. Modelling and analysing software in mCRL2. In *FACS*, volume 12018 of *Lecture Notes in Computer Science*, pages 25–48. Springer, 2019. doi:10.1007/978-3-030-40914-2_2.

9 Ramon Janssen, Frits W. Vaandrager, and Jan Tretmans. Relating alternating relations for conformance and refinement. In *IFM*, volume 11918 of *Lecture Notes in Computer Science*, pages 246–264. Springer, 2019. doi:10.1007/978-3-030-34968-4_14.

10 Maurice Laveaux, Jan Friso Groote, and Tim A. C. Willemse. Correct and efficient antichain algorithms for refinement checking. *Log. Methods Comput. Sci.*, 17(1), 2021. doi:10.23638/LMCS-17(1:8)2021.

11 Sidney C. Nogueira, Augusto Sampaio, and Alexandre Mota. Test generation from state based use case models. *Formal Aspects Comput.*, 26(3):441–490, 2014. doi:10.1007/s00165-012-0258-z.

12 Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Synchronizing asynchronous conformance testing. In *SEFM*, volume 7041 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2011. doi:10.1007/978-3-642-24690-6_23.

13 Neda Noroozi, Ramtin Khosravi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Synchrony and asynchrony in conformance testing. *Softw. Syst. Model.*, 14(1):149–172, 2015. doi:10.1007/s10270-012-0302-8.

14 Jan Peleska, Wen-ling Huang, and Ana Cavalcanti. Finite complete suites for CSP refinement testing. *Sci. Comput. Program.*, 179:1–23, 2019. doi:10.1016/j.scico.2019.04.004.

15 A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010. doi:10.1007/978-1-84882-258-0.

16 Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2008. doi:10.1007/978-3-540-85778-5_18.

17 Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1999. doi:10.1007/3-540-48320-9_6.

18 Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. doi:10.1007/978-3-540-78917-8_1.

19 Jan Tretmans and Ramon Janssen. Goodbye ioco. In *A Journey from Process Algebra via Timed Automata to Model Learning*, volume 13560 of *Lecture Notes in Computer Science*, pages 491–511. Springer, 2022. doi:10.1007/978-3-031-15629-8_26.

20 Rutger van Beusekom, Bert de Jonge, Paul F. Hoogendijk, and Jan Nieuwenhuizen. Dezyne: Paving the way to practical formal software engineering. In *F-IDE@NFM*, volume 338 of *EPTCS*, pages 19–30, 2021. doi:10.4204/EPTCS.338.4.

21 Rutger van Beusekom, Jan Friso Groote, Paul F. Hoogendijk, Robert Howe, Wieger Wesselink, Rob Wieringa, and Tim A. C. Willemse. Formalising the Dezyne modelling language in mCRL2. In *FMICS-AVoCS*, volume 10471 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2017. doi:10.1007/978-3-319-67113-0_14.

22 Petra van den Bos and Mariëlle Stoelinga. Tester versus bug: A generic framework for model-based testing via games. In *GandALF*, volume 277 of *EPTCS*, pages 118–132, 2018. doi:10.4204/EPTCS.277.9.

23 Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In *FATES*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2003. doi:10.1007/978-3-540-24617-6_7.

24 Martin Weiglhofer. *Automated Software Conformance Testing.* PhD thesis, Graz University of Technology, 2009.

25 Martin Weiglhofer and Bernhard K. Aichernig. Unifying input output conformance. In *UTP*, volume 5713 of *Lecture Notes in Computer Science*, pages 181–201. Springer, 2008. doi:10.1007/978-3-642-14521-6_11.

26 Martin Weiglhofer and Franz Wotawa. Asynchronous input-output conformance testing. In *COMPSAC (1)*, pages 154–159. IEEE Computer Society, 2009. doi:10.1109/COMPSAC.2009.194.

27 Tim A. C. Willemse. Heuristics for ioco-based test-based modelling. In *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 132–147. Springer, 2006. doi:10.1007/978-3-540-70952-7_9.