Model-based Testing

Mark TIMMER, Ed BRINKSMA, and Mariëlle STOELINGA
Formal Methods and Tools, Faculty of EEMCS
University of Twente, The Netherlands
e-mail: {timmer, brinksma, marielle}@cs.utwente.nl

Abstract. This paper provides a comprehensive introduction to a framework for formal testing using labelled transition systems, based on an extension and reformulation of the ioco theory introduced by Tretmans. We introduce the underlying models needed to specify the requirements, and formalise the notion of test cases. We discuss conformance, and in particular the conformance relation ioco. For this relation we prove several interesting properties, and we provide algorithms to derive test cases (either in batches, or on the fly).

Keywords. Model-based testing, ioco theory, conformance, labelled transition systems, formal testing

1. Introduction

Testing is the most important practical technique for the validation of software systems. Moreover, even if techniques like model checking will perhaps one day lead to the automated verification of software systems, testing remains an indispensible tool to assess the correctness of the concrete physical operation of software systems on given hardware platforms and in the context of larger, embedding systems. The ultimate reliability of critical software systems that we now depend on for vital applications in everyday life (driving a car, flying a plane, transferring money, operating on patients, etc.) can only be ascertained by testing the final implementations of the hardware and software combinations involved.

In spite of the important status of testing as a tool for reliable engineering, the consideration of testing as subject for serious academic study is comparatively late in the development of computer science, i.e., since the 1990s, as before that time most studies concerning correctness were focussed on the development of theories for program and system verification. Nevertheless, nowadays there is a considerable body of knowledge concerning testing theories and tools, most notably as applications of formal methods for concurrent systems and automata theory for dynamic system properties, and the theory of abstract data types for static properties of data structures and operations on them.

The use of formal methods in the context of testing offers the instruments for addressing the following important issues:

- The unambiguous specification of models that capture the allowed behaviours of implementations under test;

- The precise definition of the criteria for conformance, i.e., the formal definition of when the behaviour of an implementation can be considered correct with respect to the specification. Such criteria are often referred to as implementation relations;
- The precise definition of relevant concepts such as test cases, test suites, test runs, the validity of tests, etc;
- A well-defined basis for the development of algorithms for the derivation of valid tests from specifications and the evaluation of test runs, and their implementation in tools for test generation, execution and evaluation.

In this paper we give a comprehensive introduction to a framework for testing based on formal modelling by labelled transition systems and theories of observable behaviour that can be traced back to the process-algebraic approach to concurrency, and process calculi such as CCS [1] and CSP [2]. What we present is essentially an extension and reformulation of the ioco theory first presented by Jan Tretmans [3,4], which applies ideas first formulated by Brinksma for synchronously communicating systems [5], to the much more practical setting of input/output systems. The work by Brinksma, in turn, was inspired by the seminal paper of De Nicola and Hennessy that first introduced a formalised notion of testing in process algebra [6].

A central concept in the ioco theory is the notion of quiescence, which characterises system states that will not produce any output response without the provision of a new input stimulus. In the setting of input/output systems one generally assumes the systems to be input-enabled: all input actions are always possible in all system states, i.e., input can never be refused. This means that an input/output system is never formally deadlocked, since one can always execute further (input) actions. In this context quiescence becomes the meaningful representation of unproductive behaviour, comparable to deadlocked behaviour in the case of synchronously communicating systems.

Particular technical elegance of the proposed framework is achieved by representing quiescence in a state by a special output action, representing the absence of 'real' outputs in that state. This allows us to model the relevant implementation relations by the inclusion relation over sets of traces of actions, including quiescence. Such sets of generalised traces then capture the relevant notion of observable behaviour.

In the following section we give an informal overview of the main ingredients of the framework.

2. An Overview of Model-based Testing

Model-based testing includes three major stages: (1) formally modelling the requirements, (2) generating test cases from the model, and (3) running these test cases against an actual system under test and evaluating the results. To do this, a conformance relation must be selected that determines under which conditions an implementation is considered to conform to its requirements. Steps (2) and (3) are often combined, leading to so-called on-the-fly test case generation methods. If these steps are performed separately, this is called batch test case derivation.

In this section we provide a general overview of all these steps, and also explain informally how they have been implemented in the ioco framework. The remainder of the paper then thoroughly explains the mathematical details of this framework.

2.1. Formally Modelling the Requirements

The first step of a model-based testing process is to model the specification of the system under test. Basically, this boils down to writing down exactly what the system is supposed to do. A formal model enables us later on to automatically generate test cases that will verify whether the system under test indeed satisfies all these requirements.

To ensure an unambiguous test process, we need unambiguous models. Several modelling formalisms can be used for this purpose, such as VDM [7], Z [8] and PROMELA [9], but most notably finite state machines (FSMs) [10,11,12] and labelled transition systems (LTSs). The first three formalisms are specification *languages*, whereas the last two are very basic *mathematical structures*. The latter describe the required behaviour of a system by specifying the allowed interactions between the system and its environment, and are used in such a way that correct behaviour simply corresponds to paths through these models. Both FSMs and LTSs consist of a set of states in which the system can be, and describe transitions between these states. For finite state machines each transition consists of exactly one input from the user and the corresponding response to be provided by the system. For labelled transition systems each transition is labelled by precisely one action, which can be either an input of the user or a response of the system.

For an extended survey of modelling languages suitable for testing, see [13].

Formal modelling in the ioco framework. The ioco framework is based on labelled transition systems, or more specifically on input-output labelled transition systems. In contrast to FSMs, these allow for a modular approach to system specification by means of the well-known parallel composition operator. This also enables easy modelling of interleavings (whereas FSMs are more suitable for specifying synchronous systems). Considering that LTSs are fundamental in formal modelling, and that many high-level specification languages have their semantics given in terms of LTSs, it comes as no surprise that this model was chosen for the ioco framework.

More precisely, the ioco test methods we introduce are based on a specific type of LTSs; quiescent labelled transition systems (QLTSs). These systems explicitly model the required absence of outputs, called *quiescence* and denoted by the action δ.

Example 2.1. Consider a very simple music player. It contains exactly two songs and is controlled by one shuffle button. The system responds to a press on this button by nondeterministically starting one of the songs. After a while the song finishes, unless the shuffle button is pressed beforehand. In that case, a new song is selected.

This system is modelled formally as the LTS in Figure 1 (states are represented by circles, the initial state by a double circle, and transitions by arrows between states).

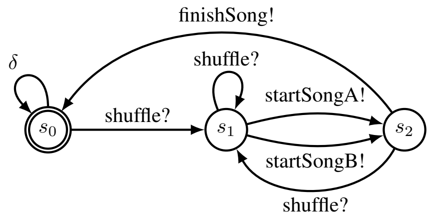


Figure 1. An LTS specifying a simple music player.

2.2. Generating Test Cases from the Model

Once a model of the specification has been made, it can be used to derive test cases for the system under test. Basically, a test case is nothing more than a sequence of inputs (stimuli) to be provided to the system and outputs expected as responses from the system. A set (or sequence) of test cases is called a test suite.

As there are infinitely many correct sequences of inputs for any nontrivial system, the test generation phase cannot produce all possible test cases. Therefore, some strategy is needed, deciding on which test cases to include into a finite test suite. Often this selection process is based on maximising some notion of *coverage* [14,15,16].

Test generation can be done manually, but preferably by a specialised test tool such a TorX [17], TGV [18], AGEDIS [19], Lutess [20,21], or TestComposer [22]. Instead of generating a batch of tests in advance, such tools might also generate tests *on the fly*; at every point in time the tool then decides whether and how to continue testing.

Test case generation in the ioco framework. Because of possible choices between different output actions (as supported by LTSs), the system under test might be allowed to respond in several different way to certain inputs. For all of these responses the test case should be able to continue testing. Therefore, in the ioco framework test cases cannot be represented by sequences of inputs and outputs anymore, but are represented as trees (or, more efficiently, as directed acyclic graphs (DAGs)).

These DAGs are again represented as LTSs. They are accompanied by an annotation function, indicating for each complete trace whether or not this course of action is allowed.

As any nontrivial LTS contains infinitely many paths, it is complicated to deal with notions of coverage in the framework of ioco. Traditionally, coverage takes a syntactic point of view, but this has several disadvantages. First of all, a different coverage figure might be assigned to systems behaving identically but being syntactically different. Second of all, the fact that some parts of a system might be more critical than others, requiring testing priorities, is not taken into account. Only a few papers discussing semantic coverage have appeared in literature [23,24], but much more research in this direction is necessary.

Example 2.2. We consider again the system specified in the previous example. A possible test scenario could be to first press the shuffle button, and then observe the output of the system. When the output is incorrect (no song is started) we immediately abort the test and fail, otherwise we observe again to check if the system finishes the song correctly.

A tree representation of the corresponding test case is provided in Figure 2. Note that, when trying to apply an input, we also take into account the possibility of an unexpected output. For lay-out purposes state names were omitted; the initial state is now indicated by an incoming arrow without source.

2.3. Running Test Cases against a System under Test

Once a batch of test cases has been generated, it should be executed against the system under test. Basically, the inputs specified by the test cases are provided to the system under test, after which the responses are logged. Clearly, this can easily be automated and performed by a test tool. Once the responses have been observed, they can be compared

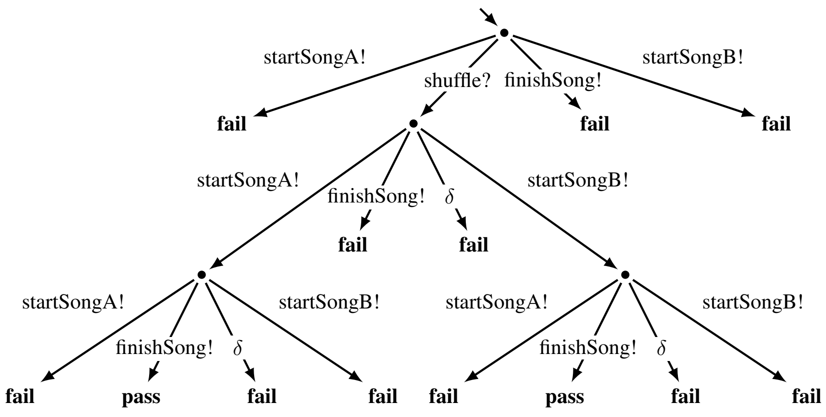


Figure 2. A test case for the music player.

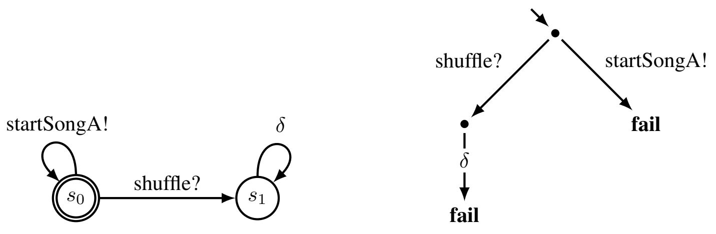
to the expected responses. When all responses are correct the system *passes* the test, otherwise it *fails*.

Test cases may either be interrupted upon detection of a failure, or continued to find more than one erroneous response.

Running test cases in the ioco framework. As both specification and test case are modelled as LTSs in the ioco framework, we can model the execution of a test case against an implementation by putting them in parallel (and synchronising on all actions). This parallel composition contains all traces that *might* be observed during the actual execution of the test case in practice. When executing a test case several times, hopefully a complete view of the parallel composition is obtained.

Example 2.3. Still considering the music player of the previous examples, Figure 3(a) shows a possible (erroneous) implementation. Note that the implementation contains two obvious mistakes: (1) the first song might start without even pressing the button, and (2) after pressing the button nothing happens anymore.

Given this implementation and the test case of the previous example, Figure 3(b) shows the test execution. Note that the parallel composition shows both errors. Indeed, when executing this test case either the song already erroneously starts before we had the chance to press the button, or we will press the button and observe nothing. During a



(a) An implementation (b) The test execution

Figure 3. An erroneous implementation of the music player and the corresponding test execution.

single execution of the test case, obviously, only one of these errors will be noticed. So, even under fairness assumptions, we will need several runs to detect all errors.

2.4. The Conformance Relation

So far we just assumed that we could make test cases and decide when responses are correct or not. To do this more precisely, a conformance relation must be defined. Such a relation exactly prescribes under what conditions an implementation conforms to a specification. For instance, when a specification prescribes two possible outputs after a certain input, the conformance relation might allow implementations to only be able to provide one of these outputs, but it might prohibit implementations that do not provide any response.

Based on the conformance relation, we can decide whether or not a test suite is sound. That is, does every implementation that conforms to its specification indeed pass the test suite? Moreover, we can talk about completeness: does every nonconforming implementation fail the test suite? Clearly, all nontrivial systems require an infinite test suite before completeness is achieved. The least that could be expected from an incomplete test suite is that it is consistent; besides passing every correct implementation, it should also fail every implementation of which it observes erroneous behaviour.

Although it might seem trivial that tests should be sound and consistent, in everyday practice many erroneous test suites are produced manually. It is therefore often said that testers are no better at writing test suites than programmers are at writing code. However, when using model-based testing, a sound test suite can be generated automatically based on a model and a conformance relation.

Conformance in the ioco framework. In the ioco framework, the conformance relation that is used is called ioco (hence the framework's name). We say that an implementation I ioco-conforms a specification S (denoted by $I \sqsubseteq_{\text{ioco}} S$) when at any point in execution it can handle at least as many inputs as the specification, and at most as many outputs. The one exception to this rule is that it is not allowed to be quiescent (i.e., not provide any output) when the specification prescribes at least one possible output.

Example 2.4. Based on these ideas, it is clear that a music player always choosing song A after a press of the shuffle button ioco-conforms the specification provided earlier (as it is allowed to provide less outputs). However, an implementation that does not play at all is not allowed (unexpected quiescence), neither is an implementation that plays a song before the button is pressed (as this would imply that more outputs are provided than allowed by the specification).

2.5. Overview of the Paper

In Section 3 we provide some basic preliminaries, which are used in Section 4 to formally introduce labelled transition systems, as well as the extension to quiescent labelled transition systems.

In Section 5 we formally define test cases. Also, we introduce annotated test cases, providing a method for denoting when a test case passes and when it fails. Based on this notion, we discuss when implementations pass or fail. Finally, we introduce conformance relations, and relate annotated test cases to such relations by means of the notions of soundness, completeness and consistency.

In Section 6 we introduce the conformance relation ioco, forming the basis of our framework. We show how it can be used to annotate test cases, and prove that this annotation is sound. Also, we provide a characterisation of completeness, and provide some other interesting properties of this conformance relation.

In Section 7 we provide an algorithm showing how a batch of test cases based on ioco can be generated. We prove that it is in principle complete. Section 8 provides a variation of this algorithm, deriving test cases on the fly.

In Section 9 we illustrate the practical applications of the ioco framework based on some tools and industrial case studies. The paper ends with related work and conclusions in Section 10.

An appendix is provided, containing proofs for all our propositions and theorems.

3. Formal Preliminaries

Given a set L, the set of all sequences over L is denoted by L^*, and the set of all nonempty sequences by L^+. Given a sequence $\sigma = a_1 a_2 \dots a_k$, we use $|\sigma| = k$ to denote its length. If $\sigma, \rho \in L^*$, then σ is a *prefix* of ρ (denoted by $\sigma \sqsubseteq \rho$) if there is a $\sigma' \in L^*$ such that $\sigma\sigma' = \rho$. If $\sigma' \in L^+$, then σ is a *proper prefix* of ρ (denoted by $\sigma \sqsubset \rho$). We denote the empty sequence by ϵ.

Given a set $S \subseteq L^*$ of sequences over L, an element $\sigma \in S$ is *maximal* with respect to $\sqsubseteq$ if there does not exist a sequence $\rho \in S$ such that $\sigma \sqsubset \rho$. Given a sequence σ we use $\sigma \setminus \{a_1, a_2, \dots, a_n\}$ to denote the sequence ρ obtained by removing every occurrence of the actions $a_1, a_2, \dots, a_n$ from σ. We lift this definition to sets of sequences in the obvious way by stating that $S \setminus \{a_1, a_2, \dots, a_n\} = \{\sigma \setminus \{a_1, a_2, \dots, a_n\} \mid \sigma \in S\}$.

We use $\mathcal{P}(L)$ to denote the powerset of L, i.e., the set of all its subsets.

4. The Underlying Models

4.1. Labelled Transition Systems (LTSs)

Definition 4.1. A *labelled transition system (LTS)* is a tuple $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$, where

- S is a set of states;
- S^0 is a nonempty set of initial states;
- $L = L_\mathrm{I} \cup L_\mathrm{O}$ is a set of labels (representing actions), partitioned into a set of input labels L_I and a set of output labels L_O (so $L_\mathrm{I} \cap L_\mathrm{O} = \emptyset$). We assume that $\tau \notin L$ and write $L_\tau = L \cup \{\tau\}$, where τ represents a silent (invisible) action. We suffix input labels by a question mark and output labels by an exclamation mark[1]. We will use the words *action* and *label* as synonyms;
- $\rightarrow \subseteq S \times L_\tau \times S$ is the transition relation. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$, $s \xrightarrow{a}$ if there exists a state $s' \in S$ such that $s \xrightarrow{a} s'$, and $s \not\xrightarrow{a}$ otherwise.

Note that S, S^0 and L are allowed to be uncountable.

We say that $\mathcal{A}$ is *input-enabled* if $s \xrightarrow{a?}$ for all $s \in S, a? \in L_\mathrm{I}$. We say that $\mathcal{A}$ is *deterministic* if $s \xrightarrow{a} s'$ implies that $a \neq \tau$, and $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ imply that $s' = s''$.

[1]Note that ? and ! are not part of the label; they only remind us which kind of action we are dealing with.

We introduce the familiar language-theoretic concepts for LTSs. As usual, the trace semantics of an LTS $\mathcal{A}$ is given by its set of traces $\mathit{traces}_{\mathcal{A}}$; that is, every trace $\sigma \in \mathit{traces}_{\mathcal{A}}$ represents correct sequential behaviour of the system modelled by $\mathcal{A}$, whereas every trace $\sigma \in L^* \setminus \mathit{traces}_{\mathcal{A}}$ represents incorrect behaviour.

Definition 4.2. Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS, then

- A *path* in $\mathcal{A}$ is a finite sequence $\pi = s_0 a_1 s_1 \dots s_n$ such that, for all $1 \leq i \leq n$, we have $s_{i-1} \xrightarrow{a_i} s_i$. When $s_0 \in S^0$ we call π an *initial path*. We denote by $\mathit{first}(\pi) = s_0$ the first state of π and by $\mathit{last}(\pi) = s_n$ the last state of π. Finally, we denote by $\mathit{paths}_{\mathcal{A}}$ the set of all paths in $\mathcal{A}$, and by $\mathit{initpaths}_{\mathcal{A}}$ the set of all initial paths in $\mathcal{A}$.
- The *trace* of a path π, $\mathit{trace}(\pi)$, is the sequence of actions that arises by removing all states s_i and all τ-actions from π. We write $\mathit{traces}_{\mathcal{A}} = \{\mathit{trace}(\pi) \mid \pi \in \mathit{initpaths}_{\mathcal{A}}\}$ for the set of all traces corresponding to initial paths in $\mathcal{A}$.
- Let $\sigma \in L^*$ be a sequence of actions and let $s, s' \in S$ be states in $\mathcal{A}$. Then, we write $s \xRightarrow{\sigma} s'$ if there exists a path $\pi \in \mathit{paths}_{\mathcal{A}}$ such that $\mathit{first}(\pi) = s$, $\mathit{trace}(\pi) = \sigma$ and $\mathit{last}(\pi) = s'$. We write $s \xRightarrow{\sigma}$ if $s \xRightarrow{\sigma} s'$ for some state $s' \in S$, and $s \not\xRightarrow{\sigma}$ otherwise.
- We use $\mathit{ctraces}_{\mathcal{A}}$ to denote the set all complete traces of $\mathcal{A}$, i.e., $\mathit{ctraces}_{\mathcal{A}} = \{\mathit{trace}(\pi) \mid \pi \in \mathit{initpaths}_{\mathcal{A}}, \nexists a \in L \, . \, \mathit{last}(\pi) \xRightarrow{a}\}$.
- We write $\mathit{reach}_{\mathcal{A}}(S', \sigma)$ for the set of states reachable from a state in $S' \subseteq S$ via $\sigma \in L^*$, i.e., $\mathit{reach}_{\mathcal{A}}(S', \sigma) = \{s \in S \mid \exists s' \in S' \, . \, s' \xRightarrow{\sigma} s\}$ (note that this set contains either one or zero elements in case $\mathcal{A}$ is deterministic). We write $\mathit{reach}_{\mathcal{A}}(\sigma)$ to abbreviate $\mathit{reach}_{\mathcal{A}}(S^0, \sigma)$, and $\mathit{reach}_{\mathcal{A}}(S')$ for the set of states that are reachable from a state in S' by any trace, i.e., $\mathit{reach}_{\mathcal{A}}(S') = \bigcup_{\sigma \in L^*} \mathit{reach}_{\mathcal{A}}(S', \sigma)$. We write $\mathit{reach}_{\mathcal{A}}$ for the set of states in $\mathcal{A}$ that are reachable from an initial state, i.e., $\mathit{reach}_{\mathcal{A}} = \mathit{reach}_{\mathcal{A}}(S^0)$.
- We write $\mathit{after}_{\mathcal{A}}(s)$ for the set of actions that are enabled from state s, i.e., $\mathit{after}_{\mathcal{A}}(s) = \{a \in L \mid s \xRightarrow{a}\}$. We lift this definition to traces by defining $\mathit{after}_{\mathcal{A}}(\sigma) = \bigcup_{s \in \mathit{reach}_{\mathcal{A}}(\sigma)} \mathit{after}_{\mathcal{A}}(s)$.

We leave out the subscript $\mathcal{A}$ from our notations if it is clear from the context.

A well-known fact from automaton theory is that every nondeterministic LTS can be transformed into a deterministic one: its determinisation [25].

Definition 4.3. Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS. Then, the *determinisation* of $\mathcal{A}$ is the LTS $\mathit{det}(\mathcal{A})$ given by

$$\mathit{det}(\mathcal{A}) = \langle \mathcal{P}(S) \setminus \{\emptyset\}, \{S^0\}, L, \rightarrow' \rangle,$$

where $\rightarrow'$ consist of all tuples (S', a, S'') with $S' \in \mathcal{P}(S) \setminus \{\emptyset\}$ and $S'' = \{s'' \in S \mid \exists s' \in S' \, . \, s' \xRightarrow{a} s''\}$ such that $S'' \neq \emptyset$.

Proposition 4.4. *Let $\mathcal{A}$ be a (possibly nondeterministic) LTS. Then, $\mathit{det}(\mathcal{A})$ is a deterministic LTS, and*

$$\mathit{traces}_{\mathcal{A}} = \mathit{traces}_{\mathit{det}(\mathcal{A})}$$

Sometimes, it can be of use to *hide* some actions of an LTS; effectively, this is the same as renaming labels to τ.

Definition 4.5. Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS and H a set of labels, then $\mathcal{A} \setminus H$ denotes the LTS

$$\mathcal{A}' = \langle S, S^0, L \setminus H, \rightarrow' \rangle,$$

where $\rightarrow'$ is the set

$$\{(s, a, s') \in \rightarrow \mid a \notin H\} \cup \{(s, \tau, s') \in S \times \{\tau\} \times S \mid \exists a \in H \ . \ (s, a, s') \in \rightarrow\}$$

Another important operator is the parallel composition operator $||$. It is used to combine two LTSs, letting them run in parallel. Parallel composition requires the two components to synchronise on their shared actions, and allows the other actions (and the unobservable action τ) to happen unsynchronised.

Definition 4.6. Let $A = \langle S_1, S_1^0, L_1, \rightarrow_1 \rangle$ and $B = \langle S_2, S_2^0, L_2, \rightarrow_2 \rangle$ be two LTSs. Then $A \,||\, B = \langle S_1 \times S_2, S_1^0 \times S_2^0, L_1 \cup L_2, \rightarrow \rangle$, with

$$\begin{aligned}
\rightarrow = \; & \{((s,t), a, (s',t')) \mid (s, a, s') \in \rightarrow_1, (t, a, t') \in \rightarrow_2, a \neq \tau\} \\
& \cup \{((s,t), a, (s',t)) \mid (s, a, s') \in \rightarrow_1, t \in S_2, a \in (L_1 \setminus L_2) \cup \{\tau\}\} \\
& \cup \{((s,t), a, (s,t')) \mid (t, a, t') \in \rightarrow_2, s \in S_1, a \in (L_2 \setminus L_1) \cup \{\tau\}\}
\end{aligned}$$

4.2. Quiescent Labelled Transition Systems (QLTSs)

As during testing we look at the outputs provided by an implementation, it is sometimes also useful to explicitly refer to the *absence* of outputs. We follow the literature by using the term *quiescence* to denote the absence of outputs, and introduce *quiescent labelled transition systems* (*QLTSs*) to explicitly model quiescence via a special output label δ. More precisely, after the δ action no other output (except for δ itself) can be produced before an input is provided. Note that it is possible that from a state both δ and some output $a!$ ($a! \neq \delta$) are enabled. This models the fact that the output $a!$ may or may not occur, a situation arising in nondeterministic LTSs. In that case, obviously, the δ action should take the system to a state where $a!$ is not enabled anymore.

Definition 4.7. Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS with $L = L_{\mathrm{I}} \cup L_{\mathrm{O}}$, and let $s \in S$ be a state of $\mathcal{A}$. Then, s is *quiescent* when $\nexists b! \in L_{\mathrm{O}} \ . \ s \xRightarrow{b!}$.

Definition 4.8. A *QLTS* is an LTS $\mathcal{A} = \langle S, S^0, L_{\mathrm{I}} \cup L_{\mathrm{O}}^{\delta}, \rightarrow \rangle$ with a special output label $\delta \in L_{\mathrm{O}}^{\delta}$ such that if $s \xrightarrow{\delta} s'$, then $s' \xrightarrow{\delta} s'$ and s' is quiescent. The following notations are used for QLTSs:

- We use $L_{\mathrm{O}} = L_{\mathrm{O}}^{\delta} \setminus \{\delta\}$ to refer to the set of regular output labels;
- We use $\mathit{out}_{\mathcal{A}}(\sigma) = \mathit{after}_{\mathcal{A}}(\sigma) \cap L_{\mathrm{O}}^{\delta}$ for the set of output actions (possibly including δ) that might be enabled after a trace σ;
- We use $\mathcal{I}(L)$ to denote the set of all input-enabled QLTSs over a label set L.

Note that this definition implies that if a state s enables δ, then it enables an infinite sequence of δ observations.

Definition 4.9. Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS with $L = L_\mathrm{I} \cup L_\mathrm{O}$ and $\delta \notin L$, then its *underlying QLTS* $\delta(\mathcal{A})$ is the QLTS $\langle S, S^0, L \cup \{\delta\}, \rightarrow' \rangle$, where $\rightarrow' = \rightarrow \cup \{(s, \delta, s) \mid s \in S, s \text{ is quiescent}\}$.

Proposition 4.10. *Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS with $\delta \notin L$, then*

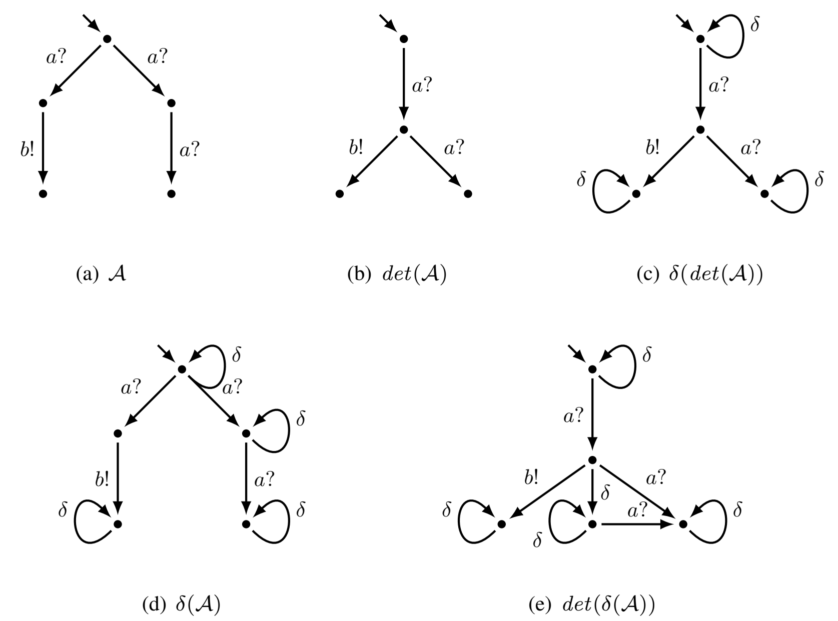
1. *the underlying QLTS $\delta(\mathcal{A})$ indeed is a QLTS;*
2. *it holds that $traces_{\delta(\mathcal{A})} \setminus \{\delta\} = traces_{\mathcal{A}}$.*

Moreover, QLTSs are closed under determinisation.

Note that this proposition implies that any LTS can be transformed to a QLTS. Without loss of generality we will therefore from now on assume that all specifications and implementations are represented as (possibly nondeterministic) QLTSs. Specifications will be referred to as $\mathcal{A}_\mathrm{s}$, and implementations as $\mathcal{A}_\mathrm{i}$. Every implementation $\mathcal{A}_\mathrm{i}$ is expected to be input-enabled. Note that in practice the behaviour of $\mathcal{A}_\mathrm{i}$ is not known a priori; the whole point of testing is finding out this behaviour and comparing it to $\mathcal{A}_\mathrm{s}$.

To transform an LTS to a deterministic QLTS one should first derive the underlying QLTS, and then determinise. The next example illustrates why doing it the other way around causes trouble.

Example 4.11. Observe the models $\mathcal{A}$, $det(\mathcal{A})$, $\delta(det(\mathcal{A}))$, $\delta(\mathcal{A})$ and $det(\delta(\mathcal{A}))$ in Figure 4. Note that $\delta(det(\mathcal{A}))$ does not capture the fact that, for instance, we might observe quiescence after providing an $a?$. Therefore, adding quiescence *after* determinisation



(a) $\mathcal{A}$ (b) $det(\mathcal{A})$ (c) $\delta(det(\mathcal{A}))$

(d) $\delta(\mathcal{A})$ (e) $det(\delta(\mathcal{A}))$

Figure 4. Determinisation and transformation to QLTS.

changes behaviour. As it is a well-known fact that determinisation preserves traces, all possible quiescence observations present in $\delta(\mathcal{A})$ are still present in $det(\delta(\mathcal{A}))$. Indeed, this model does capture the fact that we might observe quiescence after providing an $a?$.

5. Test Cases and Test Suites

5.1. Tests over an Action Signature

Testing is inherently a black-box method: to execute a test case on a given system, one only needs an executable of the implementation. We assume that each implementation is accessible via an action signature L, partitioned into a set of input actions L_{I} and a set of output actions L_{O}^{δ} (including the special action δ to denote quiescence). Test cases and test suites are now defined solely based on this action signature.

A test case t consists of a set of traces, representing the behaviour of the tester. Basically, at each moment in time the tester either provides a single input, or waits for the system to do something. This is represented by the traces in the test case. If the history of the test process is σ, and a trace $\sigma a?$ is present in t, then the tester will try to provide an input $a?$. When no such trace is present, we require the test to contain all traces of the form $\sigma b!$ with $b! \in L_{\mathrm{O}}^{\delta}$, representing the fact that the response of the system is observed. When an input is provided, the test case should also account for incoming output actions, as the implementation might be faster than the tester.

The traces in each test case t can be organised as a labelled connected directed acyclic graph, abbreviated by DAG (which can be modelled as an LTS). Clearly this DAG should not contain infinite paths (and therefore also no loops). Moreover, we require it to be deterministic, and adhere to the observations made above.

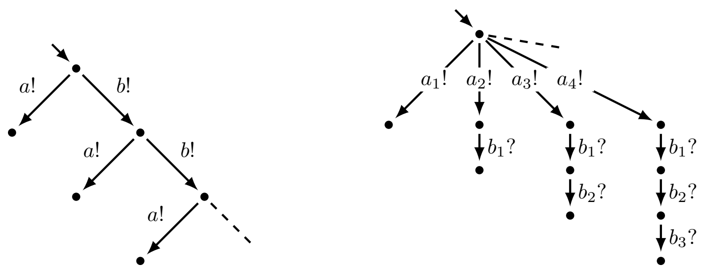
Definition 5.1.

- A *test DAG* (or shortly a *test*) over an action signature $L = L_{\mathrm{I}} \cup L_{\mathrm{O}}^{\delta}$ with $L_{\mathrm{I}} \cap L_{\mathrm{O}}^{\delta} = \emptyset$ is an LTS $\mathcal{A}$ such that

  1. $\mathcal{A}$ is deterministic and does not contain an infinite path;
  2. $\mathcal{A}$ is acyclic and connected;
  3. For every state $s \in S$, we have either *after*$(s) = \emptyset$, or *after*$(s) = L_{\mathrm{O}}^{\delta}$, or *after*$(s) = \{a?\} \cup L_{\mathrm{O}}$ for some $a? \in L_{\mathrm{I}}$.

  We denote the set of all tests over L by $\mathcal{T}(L)$.
- A *test suite* over an action signature L is a set of tests over L. We denote the set of all test suites over L by $\mathcal{TS}(L)$.
- The *depth* of a test t is the supremum of the lengths of the traces in t, i.e., it is $\sup\{|\sigma| \mid \sigma \in \textit{traces}_t\} \in \mathbb{N} \cup \{\infty\}$. We denote by $\mathcal{T}_k(L)$ the set of all tests over L of depth k.
- A test t is *linear* if there exists a trace $\sigma \in \textit{traces}_t$ such that every nonempty trace $\rho \in \textit{traces}_t$ can be written as $\sigma' a$, where $\sigma' \sqsubseteq \sigma$ and $a \in L$. The trace σ is called the *main trace* of t.

Alternatively, we can define tests as a prefix-closed set of traces. This form will turn out to be more practical when proving properties of tests.



(a) A test not allowed. (b) A test that is allowed.

Figure 5. Infinite tests.

Definition 5.2. A *test set* (or shortly a *test*) t over an action signature $L = L_\mathrm{I} \cup L_\mathrm{O}^\delta$ with $L_\mathrm{I} \cap L_\mathrm{O}^\delta = \emptyset$ is a prefix-closed subset of L^*, not containing an infinite increasing sequence $\sigma_0 \sqsubset \sigma_1 \sqsubset \sigma_2 \sqsubset \ldots$[2], and such that for all $\sigma \in t$ either

1. $\{a \in L \mid \sigma a \in t\} = \emptyset$; or
2. $\{a \in L \mid \sigma a \in t\} = L_\mathrm{O}^\delta$; or
3. $\{a \in L \mid \sigma a \in t\} = \{a?\} \cup L_\mathrm{O}$ for some $a? \in L_\mathrm{I}$.

The following proposition states that test sets and test DAGs are basically the same. Hence, we use the word "test" (or "test case") for both of them and apply terminology that applies to test DAGs (e.g., complete traces) also to test sets and vice versa.

Proposition 5.3.

1. *If $\mathcal{A}$ is a test DAG, then traces$_\mathcal{A}$ is a test set.*
2. *For every test set t, there exists a test DAG $\mathcal{A}$ such that traces$_\mathcal{A} = t$; $\mathcal{A}$ is unique upto its state names.*

Proposition 5.4. *If t is a test set and $\mathcal{A}$ its associated test DAG, then the complete traces of $\mathcal{A}$ correspond to the maximal elements of t (with respect to $\sqsubseteq$).*

Example 5.5. The restriction that a test set cannot contain an infinite increasing sequence and the restriction that a test DAG cannot contain an infinite path both make sure that every test process will eventually terminate. However, it does not mean that the size of a test set (or the depth of a test DAG) is necessarily finite.

To see this, observe the two test DAGs shown in Figure 5 (for presentation purposes not all transitions needed to make the test shown in Figure 5(b) input-enabled are drawn). The DAG shown in Figure 5(a) is not allowed, as it contains the infinite path $b!\, b!\, b!\, b! \ldots$. Therefore, it could occur that a test process based on this DAG would never end. The DAG shown in Figure 5(b), however, is a valid test. Although it has infinite depth (after all, there is no boundary below which the length of every path stays), there does not exist an infinite path; every path begins with an action a_i and then continues with $i - 1 < \infty$ actions.

Note that every test that can be obtained by cutting off Figure 5(a) at a certain depth is linear, whereas the test in Figure 5(b) is not.

[2]If L is finite, then we can replace this requirement by asking that t is finite.

Definition 5.6. Let $\mathcal{A}_\text{s} = \langle S, S^0, L, \rightarrow \rangle$ be a specification (i.e., a QLTS), then a test for $\mathcal{A}_\text{s}$ is a test over L. We denote the universe of tests and test suites for $\mathcal{A}_\text{s}$ by $\mathcal{T}(\mathcal{A}_\text{s})$ and $\mathcal{TS}(\mathcal{A}_\text{s})$, respectively.

5.2. Test Annotations, Executions and Verdicts

Before testing a system, it is obviously necessary to define which outcomes of a test case are considered correct (the system *passes* the test case), and which are considered incorrect (the system *fails* the test case). For this purpose we introduce *annotations*.

Definition 5.7. Let t be a test case, then an *annotation* of t is a function $a \colon \mathit{ctraces}_t \rightarrow \{\mathit{pass}, \mathit{fail}\}$. A pair $\hat{t} = (t, a)$ consisting of a test case together with an annotation for it is called an *annotated test case*, and a set of such pairs $\hat{T} = \{(t_i, a_i)\}$ is called an *annotated test suite*.

Running a test case can be represented as the parallel composition of the test case and the system under test. The next definition introduces the set of possible *executions* of a test case t given an implementation $\mathcal{A}_\text{i}$; all complete traces that might be observed when testing t against $\mathcal{A}_\text{i}$. Note that t and $\mathcal{A}_\text{i}$ have the same action signature, and therefore must synchronise on all actions (except possibly on τ-steps of the implementation).

Definition 5.8. Let L be an action signature, t a test case over L, and $\mathcal{A}_\text{i}$ a QLTS over L. Then $\mathit{exec}_t(\mathcal{A}_\text{i}) = \mathit{ctraces}_{t \,||\, \mathcal{A}_\text{i}}$.

Proposition 5.9. *Let L be an action signature, t a test case over L, and $\mathcal{A}_\text{i}$ a QLTS over L. Then, $\mathit{exec}_t(\mathcal{A}_\text{i}) = \mathit{ctraces}_t \cap \mathit{traces}_{\mathcal{A}_\text{i}}$.*

Note that, by only considering the complete traces of the parallel composition of t and $\mathcal{A}_\text{i}$, we discard possible test executions where $\mathcal{A}_\text{i}$ exhibits an infinite path of τ-actions. In practice, such a path would probably be ended by a time-out and considered as quiescence. However, from a theoretical perspective the system need not be quiescent, potentially resulting in an incorrect verdict. These issues can be avoided by assuming strong fairness: no infinite path of only τ-actions can be taken when there is always eventually an output action enabled.

Based on an annotated test case (or test suite) we assign a verdict to implementations; the verdict *pass* is given when the test case can never find any erroneous behaviour (i.e., there is no trace in the implementation that is also in $\mathit{ctraces}_t$ and has been annotated by *fail*), and the verdict *fail* is given otherwise.

Definition 5.10. Let L be an action signature and $\hat{t} = (t, a)$ an annotated test case over L. The *verdict function* for $\hat{t}$ is the function $v_{\hat{t}} \colon \mathcal{I}(L) \rightarrow \{\mathit{pass}, \mathit{fail}\}$, given for any input-enabled QLTS $\mathcal{A}_\text{i}$ by

$$v_{\hat{t}}(\mathcal{A}_\text{i}) = \begin{cases} \mathit{pass} & \text{if } \forall \sigma \in \mathit{exec}_t(\mathcal{A}_\text{i}) \;.\; a(\sigma) = \mathit{pass}; \\ \mathit{fail} & \text{otherwise.} \end{cases}$$

We extend $v_{\hat{t}}$ to a function $v_{\hat{T}} \colon \mathcal{I}(L) \rightarrow \{\mathit{pass}, \mathit{fail}\}$ assigning a verdict to implementations based on a test suite, by putting $v_{\hat{T}}(\mathcal{A}_\text{i}) = \mathit{pass}$ if $v_{\hat{t}}(\mathcal{A}_\text{i}) = \mathit{pass}$ for all $\hat{t} \in \hat{T}$, and $v_{\hat{T}}(\mathcal{A}_\text{i}) = \mathit{fail}$ otherwise.

Remark 5.11. Note that during (and after) testing we only have a partial view of the set $exec_t(\mathcal{A}_{\mathrm{i}})$, as we only have a partial view of $\mathcal{A}_{\mathrm{i}}$. This is one of the reasons for testing to be inherently incomplete; even though no failure has been observed, there still might be faults left in the system.

5.3. Conformance Relations, Soundness, Completeness and Consistency

Conformance relations express what it means for an implementation under test to meet a specification. Various notions of conformance exist, one of which will be defined in the next section. Formally, we define a conformance relation to be a binary relation R between QLTSs, such that, given an implementation $\mathcal{A}_{\mathrm{i}}$ and a specification $\mathcal{A}_{\mathrm{s}}$, $(\mathcal{A}_{\mathrm{i}}, \mathcal{A}_{\mathrm{s}}) \in R$ means that $\mathcal{A}_{\mathrm{i}}$ conforms to $\mathcal{A}_{\mathrm{s}}$ according to R.

Given a conformance relation, test suites can either be sound or unsound, and either complete or incomplete. Intuitively, a sound test suite never rejects a correct implementation, and a complete test suite never accepts an incorrect one.

Definition 5.12. Let R be a conformance relation, $\mathcal{A}_{\mathrm{s}}$ a specification over an action signature L, and $\hat{T}$ an annotated test suite for $\mathcal{A}_{\mathrm{s}}$. Then

- $\hat{T}$ is *sound* for $\mathcal{A}_{\mathrm{s}}$ with respect to R if for every implementation $\mathcal{A}_{\mathrm{i}} \in \mathcal{I}(L)$ it holds that $v_{\hat{T}}(\mathcal{A}_{\mathrm{i}}) = \mathit{fail} \implies (\mathcal{A}_{\mathrm{i}}, \mathcal{A}_{\mathrm{s}}) \notin R$.
- $\hat{T}$ is *complete* for $\mathcal{A}_{\mathrm{s}}$ with respect to R if for every implementation $\mathcal{A}_{\mathrm{i}} \in \mathcal{I}(L)$ it holds that $(\mathcal{A}_{\mathrm{i}}, \mathcal{A}_{\mathrm{s}}) \notin R \implies v_{\hat{T}}(\mathcal{A}_{\mathrm{i}}) = \mathit{fail}$.

Additionally, we propose a notion of *consistency*, extending soundness by requiring that implementations should not pass test suites that observe erroneous behaviour.

Definition 5.13. Let R be a conformance relation, $\mathcal{A}_{\mathrm{s}}$ a specification over an action signature L, and $\hat{t} = (t, a)$ an annotated test case for $\mathcal{A}_{\mathrm{s}}$. Then, $\hat{t}$ is *consistent* for $\mathcal{A}_{\mathrm{s}}$ with respect to R if it is sound, and for every trace $\sigma \in \mathit{ctraces}_t$ it holds that $a(\sigma) = \mathit{pass}$ implies that there exists at least one implementation containing σ that conforms to $\mathcal{A}_{\mathrm{s}}$ according to R, i.e.,

$$\forall \sigma \in \mathit{ctraces}_t \,.\, a(\sigma) = \mathit{pass} \implies \exists \mathcal{A}_{\mathrm{i}} \in \mathcal{I}(L) \,.\, \sigma \in \mathit{traces}_{\mathcal{A}_{\mathrm{i}}} \wedge (\mathcal{A}_{\mathrm{i}}, \mathcal{A}_{\mathrm{s}}) \in R.$$

An annotated test suite is consistent with respect to a conformance relation R if all its test cases are.

Obviously, for all practical purposes test suites definitely should be sound, and preferably complete (although the latter can never be achieved for any nontrivial specification and nontrivial conformance relation due to an infinite amount of possible traces). Moreover, inconsistent test suites should be avoided as they ignore erroneous behaviour.

Note that, as already mentioned in Remark 5.11, not the whole possible range of traces that $\mathcal{A}_{\mathrm{i}}$ might exhibit will in general be observed during a single test execution. Moreover, if no fairness assumption whatsoever is imposed, some behaviours might never be observed during testing. Therefore, to always eventually detect erroneous behaviour, we do not only need a complete test suite, but also some fairness assumption stating that all traces of $\mathcal{A}_{\mathrm{i}}$ will eventually be seen. And, even then, many executions of this test suite might be necessary to indeed detect all erroneous behaviour.

6. The Conformance Relation $\sqsubseteq_{\text{ioco}}$

Input-output conformance, better known as *ioco*, is an important conformance relation for QLTSs. We write $\mathcal{A}_{\text{i}} \sqsubseteq_{\text{ioco}} \mathcal{A}_{\text{s}}$ to denote that $\mathcal{A}_{\text{i}}$ conforms to $\mathcal{A}_{\text{s}}$ with respect to ioco ($\mathcal{A}_{\text{i}}$ ioco-implements $\mathcal{A}_{\text{s}}$). Basically, this is the case when $\mathcal{A}_{\text{i}}$ never provides an unexpected output when it is only fed inputs that are allowed according to $\mathcal{A}_{\text{s}}$. It should be noted that the unexpected absence of outputs, i.e., an implementation outputting nothing whereas something was expected, is also considered to be unexpected output. This immediately follows from the fact that $\delta \in L_{\text{O}}^{\delta}$ when dealing with QLTSs.

Definition 6.1. Let $\mathcal{A}_{\text{i}}, \mathcal{A}_{\text{s}}$ be QLTSs and let $\mathcal{A}_{\text{i}}$ be input-enabled. Then

$$\mathcal{A}_{\text{i}} \sqsubseteq_{\text{ioco}} \mathcal{A}_{\text{s}} \text{ if and only if } \forall \sigma \in \textit{traces}_{\mathcal{A}_{\text{s}}} \,.\, \textit{out}_{\mathcal{A}_{\text{i}}}(\sigma) \subseteq \textit{out}_{\mathcal{A}_{\text{s}}}(\sigma)$$

To test if an implementation under test conforms to a specification $\mathcal{A}_{\text{s}}$ with respect to $\sqsubseteq_{\text{ioco}}$, we apply the framework of annotated test cases and verdicts defined above. The annotation function for every test case t will be derived directly from $\mathcal{A}_{\text{s}}$; it will be denoted by $a^{\text{ioco}}_{\mathcal{A}_{\text{s}},t}$.

The basic idea is that we emit a *fail* verdict only to sequences σ that can be written as $\sigma = \sigma_1 a! \sigma_2$ such that $\sigma_1 \in \textit{traces}_{\mathcal{A}_{\text{s}}}$ and $\sigma_1 a! \notin \textit{traces}_{\mathcal{A}_{\text{s}}}$. That is, when there is an output action that leads us out of the traces of $\mathcal{A}_{\text{s}}$. Note that if we can write $\sigma = \sigma_1 b? \sigma_2$ such that $\sigma_1 \in \textit{traces}_{\mathcal{A}_{\text{s}}}$ and $\sigma_1 b? \notin \textit{traces}_{\mathcal{A}_{\text{s}}}$, then we emit a $\textit{pass}$, because in this case an unexpected input $b? \in L_{\text{I}}$ was provided by the test case. Hence, any behaviour that comes after this input is ioco-conforming.

Definition 6.2. Let t be a test case for a specification $\mathcal{A}_{\text{s}}$. The annotation function $a^{\text{ioco}}_{\mathcal{A}_{\text{s}},t} \colon \textit{ctraces}_t \to \{\textit{pass}, \textit{fail}\}$ for t is given by

$$a^{\text{ioco}}_{\mathcal{A}_{\text{s}},t}(\sigma) = \begin{cases} \textit{fail} & \text{if } \exists \sigma_1 \in \textit{traces}_{\mathcal{A}_{\text{s}}}, a! \in L_{\text{O}}^{\delta} \,.\, \sigma \sqsupseteq \sigma_1 a! \wedge \sigma_1 a! \notin \textit{traces}_{\mathcal{A}_{\text{s}}}; \\ \textit{pass} & \text{otherwise.} \end{cases}$$

6.1. Soundness, Completeness and Consistency

We now prove that given a specification $\mathcal{A}_{\text{s}}$, any test case t annotated according to $a^{\text{ioco}}_{\mathcal{A}_{\text{s}},t}$ is sound for $\mathcal{A}_{\text{s}}$ with respect to $\sqsubseteq_{\text{ioco}}$.

Proposition 6.3. *Let $\mathcal{A}_{\text{s}}$ be a specification, then the annotated test suite $\hat{T} = \{(t, a^{\textit{ioco}}_{\mathcal{A}_{\text{s}},t}) \mid t \in \mathcal{T}(\mathcal{A}_{\text{s}})\}$ is sound for $\mathcal{A}_{\text{s}}$ with respect to $\sqsubseteq_{\text{ioco}}$.*

Note that this set contains all possible test cases for $\mathcal{A}_{\text{s}}$. Thus, this set is maximal in some sense.

To prove a completeness property we first introduce a canonical form for sequences.

Definition 6.4. Let σ be a sequence over a label set L with $\delta \in L$, then its canonical form $\textit{canon}(\sigma)$ is the sequence obtained by replacing every occurring of two or more consecutive δ actions by δ, and, when σ ends in one or more δ actions, removing all those. The canonical form of a set of sequences $S \subseteq L^*$ is the set $\textit{canon}(S) = \{\textit{canon}(\sigma) \mid \sigma \in S\}$.

Proposition 6.5. *Let $\hat{T} \subseteq \{(t, a^{ioco}_{\mathcal{A}_s,t}) \mid t \in \mathcal{T}(\mathcal{A}_s)\}$ be a test suite for a specification $\mathcal{A}_s$, then*

$$\hat{T} \textit{ is complete for } \mathcal{A}_s \textit{ with respect to } \sqsubseteq_{\text{ioco}}$$
$$\Leftrightarrow$$
$$\forall \sigma \in \mathit{canon}(\mathit{traces}_{\mathcal{A}_s}) \; . \; \big(\mathit{out}_{\mathcal{A}_s}(\sigma) \neq L^{\delta}_{\text{O}} \implies \exists (t, a) \in \hat{T} \; . \; \sigma\delta \in t\big)$$

Besides being sound and possibly complete, the test cases annotated according to a^{ioco} are also consistent.

Proposition 6.6. *Let $\mathcal{A}_s$ be a specification, then the annotated test suite $\hat{T} = \{(t, a^{ioco}_{\mathcal{A}_s,t}) \mid t \in \mathcal{T}(\mathcal{A}_s)\}$ is consistent for $\mathcal{A}_s$ with respect to $\sqsubseteq_{\text{ioco}}$.*

6.2. Optimisation: Fail-fast and Input-minimal Tests

The tests from Tretmans' ioco theory [4] are required to be *fail-fast* (i.e., they stop testing after the first observation of an error) and *input-minimal* (i.e., they do not apply input actions that are unexpected according to the specification).

Definition 6.7. Let $\mathcal{A}_s$ be a specification over an action signature L, then

- a test t is *fail-fast* with respect to $\mathcal{A}_s$ if $\sigma \notin \mathit{traces}_{\mathcal{A}_s}$ implies that $\forall a \in L \; . \; \sigma a \notin t$;
- a test t is *input-minimal* with respect to $\mathcal{A}_s$ if for all $\sigma a? \in t$ with $a? \in L_{\text{I}}$ it holds that $\sigma \in \mathit{traces}_{\mathcal{A}_s}$ implies $\sigma a? \in \mathit{traces}_{\mathcal{A}_s}$.

The reason for restricting to fail-fast test cases is that ioco defines an implementation to be nonconforming if at least one nonconforming trace exists; therefore, once such a trace has been observed the verdict can be given and there is no need to continue testing. The reason for restricting to input-minimal test cases is that ioco allows any behaviour after a trace $\sigma \notin \mathit{traces}_{\mathcal{A}_s}$ anyway, invalidating the need to test for this behaviour.

Note that for a test case t that is both fail-fast and input-minimal $\sigma a? \in t$ implies $\sigma a? \in \mathit{traces}_{\mathcal{A}_s}$.

6.3. A Characterisation of $\sqsubseteq_{\text{ioco}}$ and some Properties

We prove a characterisation of $\sqsubseteq_{\text{ioco}}$ in terms of the traces of the implementation.

Theorem 6.8. *Let $\mathcal{A}_s$ be a specification and $\mathcal{A}_i$ an implementation of $\mathcal{A}_s$. Then, $\mathcal{A}_i \sqsubseteq_{\text{ioco}} \mathcal{A}_s$ if and only if for every trace $\sigma \in \mathit{traces}_{\mathcal{A}_i}$ it holds that*

$$\sigma \notin \mathit{traces}_{\mathcal{A}_s} \implies \exists \sigma' \in \mathit{traces}_{\mathcal{A}_s}, a? \in L_{\text{I}} \; . \; \sigma' a? \sqsubseteq \sigma \wedge \sigma' a? \notin \mathit{traces}_{\mathcal{A}_s}$$

An immediate result of this theorem is that ioco conformance coincides with trace inclusion in case not only the implementation, but also the specification is input-enabled.

Corollary 6.9. *Let $\mathcal{A}_s$ be an input-enabled specification and $\mathcal{A}_i$ an implementation of $\mathcal{A}_s$. Then, $\mathcal{A}_i \sqsubseteq_{\text{ioco}} \mathcal{A}_s \Leftrightarrow \mathit{traces}_{\mathcal{A}_i} \subseteq \mathit{traces}_{\mathcal{A}_s}$.*

Note that the $\Leftarrow$ direction of the corollary above (and therefore also of the theorem) only holds because $\mathcal{A}_\mathrm{s}$ and $\mathcal{A}_\mathrm{i}$ are already represented as QLTSs; trace inclusion of the LTSs $\mathcal{A}'_\mathrm{i}$ and $\mathcal{A}'_\mathrm{s}$ from which these QLTSs might have been generated does not necessarily imply that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$. The following example illustrates this.

Example 6.10. Let $\mathcal{A}'_\mathrm{s}$ be an LTS over the action signature $L = \{a?\} \cup \{b!\}$. It consists of only one state which has two self-loops: one labelled by the input action $a?$, and one labelled by the output action $b!$. Let $\mathcal{A}'_\mathrm{i}$ be an implementation of $\mathcal{A}'_\mathrm{s}$ consisting also of one state, but having only the $a?$ transition. Clearly both are input-enabled, and clearly $\mathit{traces}_{\mathcal{A}'_\mathrm{i}} \subseteq \mathit{traces}_{\mathcal{A}'_\mathrm{s}}$. However, when looking at the underlying QLTSs $\mathcal{A}_\mathrm{i}$ and $\mathcal{A}_\mathrm{s}$, we see that $\delta \in \mathit{out}_{\mathcal{A}_\mathrm{i}}(\epsilon)$, but $\delta \notin \mathit{out}_{\mathcal{A}_\mathrm{s}}(\epsilon)$. Therefore, $\mathit{out}_{\mathcal{A}_\mathrm{i}}(\epsilon) \not\subseteq \mathit{out}_{\mathcal{A}_\mathrm{s}}(\epsilon)$, and as $\epsilon \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ by definition $\mathcal{A}_\mathrm{i} \not\sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$.

The next proposition states an interesting property of $\sqsubseteq_\mathrm{ioco}$ that can easily be proven using the characterisation provided above: $\sqsubseteq_\mathrm{ioco}$ is transitive under some input-enabledness restriction. (This restriction is needed as implementations can only be ioco-correct if they are input-enabled.)

Proposition 6.11. *Let $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ be QLTSs such that $\mathcal{A}$ and $\mathcal{B}$ are input-enabled, then*

$$\mathcal{A} \sqsubseteq_\mathrm{ioco} \mathcal{B} \wedge \mathcal{B} \sqsubseteq_\mathrm{ioco} \mathcal{C} \Rightarrow \mathcal{A} \sqsubseteq_\mathrm{ioco} \mathcal{C}$$

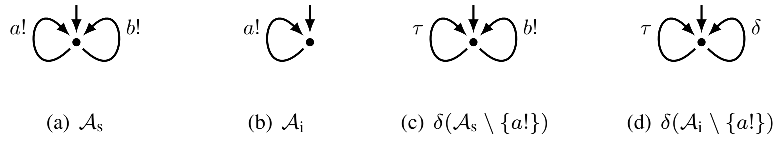
We remark that hiding does not necessarily preserve $\sqsubseteq_\mathrm{ioco}$. (Note also that quiescence might need to be re-added after hiding.)

Remark 6.12. Let $\mathcal{A}_\mathrm{s}$ be a specification over the action signature L and $\mathcal{A}_\mathrm{i}$ an implementation of $\mathcal{A}_\mathrm{s}$ such that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$, and let $H \subseteq L$. Then, not necessarily $\delta(\mathcal{A}_\mathrm{i} \setminus H) \sqsubseteq_\mathrm{ioco} \delta(\mathcal{A}_\mathrm{s} \setminus H)$.

To see why this is the case, let $\mathcal{A}_\mathrm{s}$ and $\mathcal{A}_\mathrm{i}$ be given as in Figure 6(a) and 6(b) (and assume that $L_\mathrm{I} = \emptyset$). As $\mathcal{A}_\mathrm{i}$ and $\mathcal{A}_\mathrm{s}$ are both input-enabled and $\mathit{traces}_{\mathcal{A}_\mathrm{i}} \subseteq \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, we obtain $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$ using Corollary 6.9. However, after hiding $a!$ and re-adding quiescence we get the QLTSs shown in Figure 6(c) and 6(d). Now, $\delta(\mathcal{A}_\mathrm{i} \setminus \{a!\}) \not\sqsubseteq_\mathrm{ioco} \delta(\mathcal{A}_\mathrm{s} \setminus \{a!\})$, as $\epsilon \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, and $\mathit{out}_{\delta(\mathcal{A}_\mathrm{i} \setminus \{a!\})}(\epsilon) = \{\delta\} \not\subseteq \{b!\} = \mathit{out}_{\delta(\mathcal{A}_\mathrm{s} \setminus \{a!\})}(\epsilon)$.

7. Batch Test Case Derivation for $\sqsubseteq_\mathrm{ioco}$

We so far defined a framework in which specifications can be modelled as QLTSs and test cases for them can be specified, annotated and executed. Moreover, we presented the conformance relation ioco, and provided a way to annotate test cases according to ioco



(a) $\mathcal{A}_\mathrm{s}$ (b) $\mathcal{A}_\mathrm{i}$ (c) $\delta(\mathcal{A}_\mathrm{s} \setminus \{a!\})$ (d) $\delta(\mathcal{A}_\mathrm{i} \setminus \{a!\})$

Figure 6. Illustration of Remark 6.12.

Algorithm 1: Batch test case generation for ioco.

```
Input: A specification $\mathcal{A}_s$ and a history $\sigma \in traces_{\mathcal{A}_s}$
Output: A test case $t$ for $\mathcal{A}_s$ such that $t$ is input-minimal and fail-fast

procedure batchGen($\mathcal{A}_s, \sigma$)
  [true] $\rightarrow$
      return $\{\epsilon\}$
  [true] $\rightarrow$
      result := $\{\epsilon\}$
      forall $b! \in L_O^\delta$ do
        if $\sigma b! \in traces_{\mathcal{A}_s}$ then
            result := result $\cup \{b!\sigma' \mid \sigma' \in$ batchGen$(\mathcal{A}_s, \sigma b!)\}$
        else
            result := result $\cup \{b!\}$
        end
      end
      return result
  [$\sigma a? \in traces_{\mathcal{A}_s}$] $\rightarrow$
      result := $\{\epsilon\} \cup \{a?\sigma' \mid \sigma' \in$ batchGen$(\mathcal{A}_s, \sigma a?)\}$
      forall $b! \in L_O$ do
        if $\sigma b! \in traces_{\mathcal{A}_s}$ then
            result := result $\cup \{b!\sigma' \mid \sigma' \in$ batchGen$(\mathcal{A}_s, \sigma b!)\}$
        else
            result := result $\cup \{b!\}$
        end
      end
      return result
```

in a sound manner. Finally, we discussed that we can restrict test suites to only contain fail-fast and input-minimal test cases.

The one thing still missing is a procedure to automatically generate test cases from a specification. This is accomplished by the function batchGen, captured by Algorithm 1. The input of the function is a specification $\mathcal{A}_s$ and a history $\sigma \in traces_{\mathcal{A}_s}$. The output then is a test case that can be applied *after* the history σ has taken place. The idea is to call the function initially with history ϵ, that way obtaining a test case that can be applied without any start-up phase.

When the initial call to batchGen is done, a nondeterministic choice is made. Either the empty test case is returned, a test case is generated that starts by observation, or a test case is generated that starts by stimulation. Stimulation is only possible when there is at least one input action allowed by the specification; without this guard the resulting test case would not necessarily become input-minimal.

In case stimulation of some input action $a?$ is chosen, this results in the test case containing the empty trace ϵ (to stay prefix-closed), a number of traces of the form $a?\sigma'$ where σ' is a trace from a test case starting with history $\sigma a?$, and, for every possible output action $b! \in L_O$ (so $b! \neq \delta$), a number of traces of the form $b!\sigma'$, where σ' is a trace from a test case starting with history $\sigma b!$. If the output $b!$ is erroneous, only the trace $b!$ is added to make sure that the resulting test case will be fail-fast.

In case observation is chosen, this results in the test case containing the empty trace ϵ (again, to stay prefix-closed) and, for every possible output action $b! \in L_{\mathrm{O}}^{\delta}$, a number of traces of the form $b!\sigma'$, where σ' is a trace from a test case starting with history $\sigma b!$. Again, we immediately stop after an erroneous output.

Remark 7.1. Note that, for efficiency reasons, the algorithm could be changed to remember the states in which the system might be after history σ. Then, the parameters of batchGen would become $(\mathcal{A}_{\mathrm{s}}, \sigma, S')$, the conditions in line 6 and 13 would become $\exists s \in S' \ . \ b! \in \mathit{after}_{\mathcal{A}_{\mathrm{s}}}(s)$, the condition in line 10 would become $\exists s \in S' \ . \ a? \in \mathit{after}_{\mathcal{A}_{\mathrm{s}}}(s)$, the recursive calls in line 7 and 14 would add a third parameter $\mathit{reach}_{\mathcal{A}_{\mathrm{s}}}(S', b!)$, and the recursive call in line 11 would add a third parameter $\mathit{reach}_{\mathcal{A}_{\mathrm{s}}}(S', a?)$.

Remark 7.2. Clearly, it is impossible to explicitly store any nontrivial test case for a specification over an infinite number of actions, as for such systems a single observation already leads to an infinite test case. In that case, the algorithm should be considered a pseudo-algorithm. The algorithm for on-the-fly test case derivation, presented in the next section, will still be feasible.

Theorem 7.3. *Let $\mathcal{A}_{\mathrm{s}}$ be a specification, and $t = \mathrm{batchGen}(\mathcal{A}_{\mathrm{s}}, \epsilon)$. Then, t is a fail-fast and input-minimal test case for $\mathcal{A}_{\mathrm{s}}$.*

Note that it follows from Proposition 6.3 that $\hat{t} = (t, a^{\mathrm{ioco}}_{\mathcal{A}_{\mathrm{s}},t})$ is sound for $\mathcal{A}_{\mathrm{s}}$ with respect to $\sqsubseteq_{\mathrm{ioco}}$, for any test case t produced by the algorithm.

The next theorem states that, in principe, every possible fault can be discovered by a test case generated using Algorithm 1. More specifically even, it can always be discovered by a *linear* test case.

Theorem 7.4. *Let $\mathcal{A}_{\mathrm{s}}$ be a specification, and T the set of all linear test cases that can be generated using Algorithm 1. Then, the annotated test suite $\hat{T} = \{(t, a^{ioco}_{\mathcal{A}_{\mathrm{s}},t}) \mid t \in T\}$ is complete for $\mathcal{A}_{\mathrm{s}}$ with respect to $\sqsubseteq_{\mathrm{ioco}}$.*

The following corollary immediately follows from Theorem 7.4.

Corollary 7.5. *Let $\mathcal{A}_{\mathrm{s}}$ be a specification, and T the test suite consisting of all test cases that can be generated by Algorithm 1. Then, the annotated test suite $\hat{T} = \{(t, a^{ioco}_{\mathcal{A}_{\mathrm{s}},t}) \mid t \in T\}$ is complete for $\mathcal{A}_{\mathrm{s}}$ with respect to $\sqsubseteq_{\mathrm{ioco}}$.*

Although the set of all test cases that can be generated using the algorithm is complete, some issues need to be taken into consideration.

First of all, as mentioned before, almost every system needs an infinite test suite to be tested completely, which of course is not achievable in practice. In case of a countable number of actions and states this test suite can at least be provided by the algorithm in the limit to infinitely many recursive steps, but for uncountable specifications this would not even be the case anymore (because in infinitely many steps the algorithm is only able to provide a countable set of test cases).

Second of all, although the set of all test cases derivable using the algorithm is in theory complete, this does not mean that every erroneous implementation would be detected by running all of these tests once. After all, because of nondeterminism, erroneous behaviour might not show during testing, even though it might turn up afterwards. Therefore, we need a notion of fairness, even stronger than the one discussed in Section 5.2. It requires that, when testing a system infinitely often, every possible outcome of every nondeterministic choice is taken infinitely often. In that case, the complete test set can indeed observe all possible traces of an implementation, and no erroneous behaviour is allowed to hide until testing has finished.

Despite these restrictions, the completeness theorem provides us important information about the test derivation algorithm: it has no ‘blind spots’. That is, for every possible erroneous implementation there exists a test case that can be generated using Algorithm 1 and can detect the erroneous behaviour. So, in principle every fault can be detected.

8. On-the-fly Test Case Derivation for $\sqsubseteq_{\text{ioco}}$

Instead of executing predefined test cases, it is also possible to derive test cases on the fly. A procedure to do this in a sound manner is depicted by Algorithm 2.

The input of the algorithm consists of a specification $\mathcal{A}_{\text{s}}$, a concrete implementation I, and an upper bound $n \in \mathbb{N}$ on the test depth. The algorithm contains one local variable, σ, which represents the trace obtained thus far; it is therefore initialised to the

Algorithm 2: On-the-fly test case derivation for ioco.

Input: A specification $\mathcal{A}_{\text{s}}$, a concrete implementation I, and an upper bound $n \in \mathbb{N}$ on the test depth.
Output: The verdict *pass* when the observed behaviour of I during n test steps was ioco-conform $\mathcal{A}_{\text{s}}$, and the verdict *fail* when a nonconforming trace was observed during the test.

```
σ := ε
while $|\sigma| < n$ do
    [true] →
        observe $I$'s next output $b!$ (possibly $\delta$)
        $\sigma := \sigma b!$
        if $\sigma \notin traces_{\mathcal{A}_s}$ then return $fail$
    [$\sigma a? \in traces_{\mathcal{A}_s}$] →
        try
            atomic
                stimulate $I$ with $a?$
                $\sigma := \sigma a?$
            end
        catch an output $b!$ occurs before $a?$ could be provided
            $\sigma := \sigma b!$
            if $\sigma \notin traces_{\mathcal{A}_s}$ then return $fail$
        end
end
return $pass$
```

empty trace ϵ. Then, the while loop is executed as long as the length of σ is smaller than n. As every iteration corresponds to one test step, this makes sure that at most n test steps will be performed.

For every test step there is a nondeterministic choice between observing or stimulating the implementation by any of the input actions that are enabled given the history σ and the specification $\mathcal{A}_{\mathrm{s}}$. In case observation is chosen, the output provided by the implementation (either a real output action or δ) is appended to σ. Also, the correctness of this output is verified by checking if the trace obtained thus far is contained in $\mathit{traces}_{\mathcal{A}_{\mathrm{s}}}$. If not, the verdict *fail* can be given, otherwise we continue. In case stimulation is chosen, the implementation is stimulated with one of the inputs that are allowed by the specification, and the history is updated accordingly. By definition of ioco no *fail* verdict can immediately follow from stimulation, so we continue with the next iteration.

As the implementation might provide an output action before we are able to stimulate, a try-catch block is positioned around the stimulation to be able to handle an incoming output action. Moreover, the stimulation and the update of σ are put in an atomic block, preventing the scenario where an output that occurs directly after a stimulation prevents σ from being updated properly.

Theorem 8.1. *Algorithm 2 is sound with respect to* $\sqsubseteq_{\mathrm{ioco}}$.

Note that the algorithm is obviously not complete, as it can only test a finite number of traces. However, just as for the batch generation algorithm, it does not have any blind spots. After all, it is not difficult to see that any given erroneous trace can also be detected with the on-the-fly algorithm (under the same fairness assumptions), by resolving the nondeterministic choices in the right way.

9. Tool and Case Studies

Generating test cases and executing them against an implementation can be done manually, but obviously for large systems one wishes to automate this. Therefore, in the recent years several tools applying the ioco test framework have been developed, most notably TorX [17] (later re-released as JTorX [26]) and TGV [18]. Using such tools, many case studies have shown the practical applicability of model-based testing.

An interesting example is the formal testing of the payment box of a Dutch Highway Tolling System using TorX [27]. As this system was supposed to be used to automatically charge fees from thousands of vehicle drivers passing a toll gate on a highway each day, its correctness was of the highest importance. Because of the high amount of vehicles passing within a short amount of time, parallel transactions needed to be supported. Moreover, encryption needed to be used as electronic payments were involved. Because of this combination of speed, parallelism and encryption, testing was a complex issue [28]. After some conventional tests, the system requirements were specified formally and validated using model checking. During this step an important design error, which was not detected during conventional testing, was found. Later, during the actual testing of the system with the test tool TorX, one additional error was found.

Another example is the testing of the Dutch Oosterschelde Storm Surge Barrier. Here, TorX was used to check the control program of the barrier [29]. To deal with timely responses, the tool was extended slightly to handle timing. No errors were found, increasing the confidence in the system. More recently, an electronic passport was tested using model-based methods [30]. For this, an extension of TorX supporting symbolic test generation was used [31]. Also in this case no errors were found, although more than 100.000 protocol steps were performed (in just one night). Refinements of the model allowed the testers to investigate how some underspecified aspects of the protocols where dealt with by the system.

10. Related Work and Conclusions

As already indicated in the introduction, this work is a reformulation and extension of the original publications on ioco by Tretmans [3,4]. An important difference between our presentation and that of Tretmans is that we formulated the whole theory completely in terms of (enriched) traces of labelled transition systems without resorting to process algebraic constructs. Also, there are some subtler differences, viz.:

- Our definition of quiescent transitions has been altered slightly, such that they are preserved under determinisation of the transition systems;
- We do not need the assumption that the transition systems are strongly convergent, i.e. we do allow τ-loops in the implementations under test. In our set-up diverging test runs simply do not affect the set of completed test runs, and therefore also do not affect the test evaluations. If diverging test runs must be excluded to avoid infinite internal computations at the test execution level, one must resort to standard fairness assumptions;
- Our presentation does, in principle, allow for uncountable numbers of states and actions, for which the framework remains intact. This is only useful, however, in the presence of formalisms in which (test) processes over such uncountable sets can be effectively characterised.
- We introduced a novel notion of consistency for test suites, requiring them to fail any implementation that exhibits erroneous behaviour.

Similar work to Tretmans' and ours using a different but closely related implementation relation was published by Phalippou [32], which formalised the principles behind the testing tool TVEDA [33]. The ioco framework can be successfully generalised to real-time systems [34,35], whilst maintaining a useful notion of quiescence. In fact, the framework, including its schemata for test derivation, turns out to be quite generic so that all sorts of extensions and variants of ioco have been pioneered, such as mioco [36], uioco [37], sioco [31], and hioco [38].

There exist numerous approaches to real-time testing that are based on (timed) trace inclusion as the core implementation relation, e.g. [39,40]. Their modelling power is slightly weaker, as they can only characterise the absence of actions for a finite amount of time, but not deadlock or livelock. Consequently, they are conservative extensions of untimed testing frameworks that only address safety properties, not liveness properties, as is the case for ioco. Still, the timed trace approach works very well for many practical cases, and is supported by powerful tools, such as the testing module of the Uppaal tool set [41].

Over the years, the ioco framework has established itself as the robust core for a considerable number of theories and tools for conformance testing in different settings, and well-tested, real-life applications. This paper contains the hard core of that successful framework that represents our by now well-established understanding of the desired relation between useful implementation relations for dynamic behaviour on the one hand, and test generation and evaluation on the other hand.

References

[1] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[2] C.A.R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.

[3] G. J. Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.

[4] 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.

[5] H. Brinksma. A theory for the derivation of tests. In *Proceedings of the 8th International Workshop on Protocol Specification, Testing, and Verification (PSTV '88)*, pages 63–74, 1988.

[6] R. de Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.

[7] C.B. Jones. *Systematic software development using VDM*. Prentice Hall International (UK) Ltd., 1986.

[8] J.M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, New York, NY, USA, 1988.

[9] G.J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[10] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978.

[11] B. Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[12] A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes (MOVEP '00)*, volume 2067 of *Lecture Notes in Computer Science*, pages 196–205. Springer, 2000.

[13] A. Hartman, M. Katara, and S. Olvovsky. Choosing a test modeling language: A survey. In *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing*, volume 4383 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2006.

[14] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992.

[15] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines — a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[16] L. Nachmanson, M. Veanes, W. Schulte, N. Tillmann, and W. Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Software Engineering Notes*, 29(4):55–64, 2004.

[17] A.F.E. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L.M.G. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In *Proceedings of the IFIP TC6 12th International Workshop on Testing Communicating Systems (IWTCS '99)*, volume 147 of *IFIP Conference Proceedings*, pages 179–196. Kluwer, 1999.

[18] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[19] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *Proceedings of the 7th International Conference on UML Modeling Languages and Applications (UML '04)*, volume 3297 of *Lecture Notes in Computer Science*, pages 277–280. Springer, 2004.

[20] L. du Bousquet and N. Zuanon. An overview of Lutess: A specification-based tool for testing synchronous software. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE '99)*, pages 208–215, 1999.

[21] V. Papailiopoulou. Automatic test generation for LUSTRE/SCADE programs. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*, pages 517–520. IEEE, 2008.

[22] M. Schmitt, M. Ebner, and J. Grabowski. Test generation with AUTOLINK and TESTCOMPOSER. In *Proceedings of the 2nd Workshop on SDL and MSC (SAM '02)*, pages 218–232, 2000.

[23] L. Brandán Briones, H. Brinksma, and M. I. A. Stoelinga. A semantic framework for test coverage. In *Proceedings of the 4th International Symposium on Automated Technology for Verification and Analysis (ATVA '06)*, volume 4218 of *Lecture Notes in Computer Science*, pages 399–414. Springer, 2006.

[24] M.I.A. Stoelinga and M. Timmer. Interpreting a successful testing process: risk and actual coverage. In *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE '09)*, pages 251–258. IEEE Computer Society, 2009.

[25] T. A. Sudkamp. *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley, Boston, MA, USA, 1997.

[26] A.F.E. Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer, 2010.

[27] R.G. de Vries, A.F.E. Belinfante, and J. Feenstra. Automated testing in practice: The highway tolling system. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems (TestCom '02)*, volume 210 of *IFIP Conference Proceedings*, pages 219–234. Kluwer, 2002.

[28] G.J. Tretmans and H. Brinksma. TorX: Automated model-based testing. In *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pages 31–43, 2003.

[29] A.F.E. Belinfante. Timed testing with TorX: The Oosterschelde storm surge barrier. In *Proceedings of the 8th Dutch Testing Day*, 2002.

[30] W. Mostowski, E. Poll, J. Schmaltz, J. Tretmans, and R. Wichers Schreur. Model-Based Testing of Electronic Passports. In *Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '09)*, volume 5825 of *Lecture Notes in Computer Science*, pages 207–209. Springer, 2009.

[31] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES '04), Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2004.

[32] Phalippou. *Relations d'Implantation et Hypothèses de Test sur des Automates à Entrées et Sorties*. PhD thesis, L'Université de Bordeaux I, 1994.

[33] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking test generation with verification techniques. In *Proceedings of the 8th International Workshop on Protocol Test Systems (IWPTS '96)*, 1996.

[34] L. Brandán Briones and H. Brinksma. A test generation framework for quiescent real-time systems. In *Proceedings of the 4th International Workshop on Formal Approaches to Software Testing (FATES '04), Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 64–78. Springer, 2004.

[35] H.C. Bohnenkamp and A. Belinfante. Timed testing with TorX. In *Proceedings of the 13th International Symposium on Formal Methods*, volume 3582 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2005.

[36] A.W. Heerink. *Ins and Outs in Refusal Testing*. PhD thesis, University of Twente, The Netherlands, 1998.

[37] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software (FATES '03)*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2003.

[38] M. van Osch. Hybrid input-output conformance and test generation. In *Proceedings of the First Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV '06), Revised Selected Papers*, volume 4262 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2006.

[39] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[40] J. Springintveld, F.W. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001.

[41] G. Behrmann, A. David, K.G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems (QEST '06)*, pages 125–126, 2006.

A. Proofs

Proposition 4.10. *Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS with $\delta \notin L$, then*

1. *the underlying QLTS $\delta(\mathcal{A})$ indeed is a QLTS;*
2. *it holds that traces$_{\delta(\mathcal{A})} \setminus \{\delta\} =$ traces$_{\mathcal{A}}$.*

Moreover, QLTSs are closed under determinisation.

Proof.

1. Let $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ be an LTS with $\delta \notin L$, and $\delta(\mathcal{A}) = \langle S, S^0, L \cup \{\delta\}, \rightarrow' \rangle$ its underlying QLTS. For $\delta(\mathcal{A})$ to be a QLTS it should hold that if $s \xrightarrow{\delta} s'$, then $s' \xrightarrow{\delta} s'$ and s' is quiescent. Since $\delta \notin L$, the only δ steps present in $\delta(\mathcal{A})$ are those introduced by the transformation of $\mathcal{A}$ to $\delta(\mathcal{A})$. So, by definition, for any $s \xrightarrow{\delta} s'$ it holds that $s' = s$ (so $s' \xrightarrow{\delta} s'$), and that s' is quiescent.
2. The only difference between an LTS $\mathcal{A}$ (with $\delta \notin L$) and $\delta(\mathcal{A})$ is that $\delta(\mathcal{A})$ might contain some additional self-loops labelled δ. Therefore, every trace $\sigma \in$ *traces*$_{\mathcal{A}}$ is also in *traces*$_{\delta(\mathcal{A})}$, and as $\delta \notin L$ also $\sigma \in$ *traces*$_{\delta(\mathcal{A})} \setminus \{\delta\}$.
   Now let σ be a trace in *traces*$_{\delta(\mathcal{A})} \setminus \{\delta\}$, and let $\rho \in$ *traces*$_{\delta(\mathcal{A})}$ be one of the corresponding traces that do potentially still include δ. Let π be a path in $\delta(\mathcal{A})$ such that $trace(\pi) = \rho$. The only transitions of this path that are not in $\mathcal{A}$ are self-loops labelled δ. Therefore, the path π' obtained by omitting the δ self-loops from π is in $\mathcal{A}$ and has precisely σ as its trace, so $\sigma \in$ *traces*$_{\mathcal{A}}$.
   Hence, *traces*$_{\delta(\mathcal{A})} =$ *traces*$_{\mathcal{A}}$.
3. Let $\mathcal{Q} = \langle S, S^0, L, \rightarrow \rangle$ be a QLTS and $det(\mathcal{Q})$ its determinisation. So, $det(\mathcal{Q}) = \langle \mathcal{P}(S) \setminus \{\emptyset\}, \{S^0\}, L, \rightarrow' \rangle$, where $\rightarrow'$ consist of all tuples (S', a, S'') with $S' \subseteq S$ and $S'' = \{s'' \in S \mid \exists s' \in S' \,.\, s' \overset{a}{\Longrightarrow} s''\} \neq \emptyset$. To show that $det(\mathcal{Q})$ is still a QLTS we prove that if $S' \xrightarrow{\delta}' S''$, then $S'' \xrightarrow{\delta}' S''$ and S'' is quiescent.
   By definition $S' \xrightarrow{\delta}' S''$ implies $S'' = \{s'' \in S \mid \exists s' \in S' \,.\, s' \overset{\delta}{\Longrightarrow} s''\} \neq \emptyset$. Since every state $s'' \in S''$ was reached by a δ step, they must all be quiescent and have a transition $s'' \xrightarrow{\delta} s''$. Hence, $S'' \xrightarrow{\delta}' S''$ and S'' is quiescent. □

Proposition 5.3.

1. *If $\mathcal{A}$ is a test DAG, then traces$_{\mathcal{A}}$ is a test set.*
2. *For every test set t, there exists a test DAG $\mathcal{A}$ such that traces$_{\mathcal{A}} = t$; $\mathcal{A}$ is unique upto its state names.*

Proof.

1. Let $\mathcal{A}$ be a test DAG over an action signature $L = L_{\mathrm{I}} \cup L_{\mathrm{O}}^{\delta}$. We prove that *traces*$_{\mathcal{A}}$ is a prefix-closed subset of L^*, not containing an infinite increasing sequence $\sigma_0 \sqsubset \sigma_1 \sqsubset \sigma_2 \sqsubset \dots$, and such that for all $\sigma \in$ *traces*$_{\mathcal{A}}$ either

   (a) $\{a \in L \mid \sigma a \in \textit{traces}_{\mathcal{A}}\} = \emptyset$; or
   (b) $\{a \in L \mid \sigma a \in \textit{traces}_{\mathcal{A}}\} = L_{\mathrm{O}}^{\delta}$; or
   (c) $\{a \in L \mid \sigma a \in \textit{traces}_{\mathcal{A}}\} = \{a?\} \cup L_{\mathrm{O}}$ for some $a? \in L_{\mathrm{I}}$.

First, let $\sigma a \in$ *traces*$_{\mathcal{A}}$, then by definition there must be some initial path $\pi = s_0 a_1 s_1 \ldots a_{n-1} s_{n-1} a s_n$ in $\mathcal{A}$ such that σa is obtained by omitting all states from π. We can write $\pi = \pi' a s_n$, and obviously the initial path π' is also in $\mathcal{A}$. The corresponding trace is σ, so $\sigma \in$ *traces*$_{\mathcal{A}}$ and *traces*$_{\mathcal{A}}$ is indeed prefix-closed. Second, from the requirement that test DAGs do not contain infinite paths it immediately follows that *traces*$_{\mathcal{A}}$ does not contain an infinite increasing sequence. Third, when $\sigma \in$ *traces*$_{\mathcal{A}}$, then there must be a unique initial path $\pi = s_0 a_1 s_1 \ldots a_n s_n$ in $\mathcal{A}$ such that $trace(\pi) = \sigma$. By condition 3 of the definition of test DAGs either *after*$(s_n) = \emptyset$, or *after*$(s_n) = L_{\mathrm{O}}^{\delta}$, or *after*$(s_n) = \{a?\} \cup L_{\mathrm{O}}$ for some $a? \in L_{\mathrm{I}}$. So, by definition indeed $\{a \in L \mid \sigma a \in$ *traces*$_{\mathcal{A}}\}$ is either $\emptyset$, L_{O}^{δ}, or $\{a?\} \cup L_{\mathrm{O}}$ for some $a? \in L_{\mathrm{I}}$.

2. Let t be a test set over a signature L. We (a) construct a test DAG $\mathcal{A}$ corresponding to t, (b) prove that it is indeed a test DAG, (c) prove that *traces*$_{\mathcal{A}} = t$, and (d) prove that it is unique upto state names.

   (a) To ascertain that *traces*$_{\mathcal{A}} = t$, for every trace $\sigma \in t$ there should be a path $\pi = s_0 a_1 s_1 a_2 \ldots a_n s_n$ in $\mathcal{A}$ such that $\sigma = a_1 a_2 \ldots a_n$. We achieve this by creating exactly one state in $\mathcal{A}$ for every trace $\sigma \in t$ and adding an edge $\sigma \xrightarrow{a} \rho$ if and only if $\rho = \sigma a$. Formally, $\mathcal{A} = \langle S, S^0, L, \rightarrow \rangle$ with $S = t$, $S^0 = \epsilon$, and $\rightarrow = \{(\sigma, a, \rho) \in t \times L \times t \mid \rho = \sigma a\}$.

   (b) Clearly $\mathcal{A}$ is deterministic; when $\sigma \xrightarrow{a} \rho$ and $\sigma \xrightarrow{a} \rho'$, then $\rho = \rho'$ as they are both equal to σa. Also, $\mathcal{A}$ does not contain an infinite branch, as this could only result from an infinite increasing sequence in t. It is easy to see that $\mathcal{A}$ is a DAG (actually, it is even a tree); each state σa has only one incoming transition: from state σ with label a. The initial state ϵ does not not have any prefixes, and therefore also no incoming transitions. Also, it is connected, as there clearly is a path from ϵ to any other state. Moreover, condition 3 of the definition of test DAGs is easily seen to be satisfied as an immediate consequence of the requirements on test sets.

   (c) We now prove that *traces*$_{\mathcal{A}} = t$.
   ($\supseteq$) First, let $\sigma = a_1 a_2 a_3 \ldots a_n \in t$. As t is prefix-closed, it will contain the traces $\epsilon, a_1, a_1 a_2, a_1 a_2 a_3$, and so on. By the construction of $\mathcal{A}$ there is a state in $\mathcal{A}$ for all these traces. Moreover, there is a transition from the initial state ϵ to state a_1 labelled by the action a_1, there is a transition from a_1 to $a_1 a_2$ labelled a_2, and so on. Therefore, $\mathcal{A}$ contains the path $\pi = \overline{\epsilon} a_1 \overline{a_1} a_2 \overline{a_1 a_2} a_3 \ldots a_n \overline{\sigma}$ (with state names overlined for readability). Clearly, $trace(\pi) = \sigma$, so $\sigma \in$ *traces*$_{\mathcal{A}}$.
   ($\subseteq$) Second, let $\sigma = a_1 a_2 a_3 \ldots a_n \in$ *traces*$_{\mathcal{A}}$. Then there must be a path $\pi = s_0 a_1 s_1 a_2 s_2 a_3 \ldots a_n s_n$ in $\mathcal{A}$ with $trace(\pi) = \sigma$. By the construction of $\mathcal{A}$ we know that $s_n = \sigma$, and as all states are elements of t we have $\sigma \in t$.

   (d) We now prove that $\mathcal{A}$ is unique upto state names. As $\mathcal{A}$ is required to be deterministic (condition 1), changing any transition would alter its set of traces. Moreover, adding unreachable states would violate condition 2, since the resulting LTS would not be connected anymore. As *traces*$_{\mathcal{A}}$ should be equal to t, clearly the test DAG $\mathcal{A}$ we constructed is unique upto state names. $\square$

Proposition 5.4. *If t is a test set and $\mathcal{A}$ its associated test DAG, then the complete traces of $\mathcal{A}$ correspond to the maximal elements of t (with respect to $\sqsubseteq$).*

Proof. Let t be a test set and $\mathcal{A}$ its associated test DAG, and let σ be a maximal element of t. This implies that there does not exist an $a \in L$ such that $\sigma a \in t$. By the construction of $\mathcal{A}$ it therefore immediately follows that σ is a complete trace of $\mathcal{A}$.

Let σ be a complete trace of $\mathcal{A}$, then by the construction of $\mathcal{A}$ it follows that there does not exist a state ρ such that $\rho = \sigma a$ for some $a \in L$, so by the correspondence between states of $\mathcal{A}$ and traces in t it follows that there does not exist such a trace ρ in t. As t is prefix-closed, this implies that σ is a maximal element. □

Proposition 5.9. *Let L be an action signature, t a test case over L, and $\mathcal{A}_\text{i}$ a QLTS over L. Then, $exec_t(\mathcal{A}_\text{i}) = ctraces_t \cap traces_{\mathcal{A}_\text{i}}$.*

Proof. ($\subseteq$). Let $\sigma \in exec_t(\mathcal{A}_\text{i})$, so by definition $\sigma \in ctraces_{t \,||\, \mathcal{A}_\text{i}}$. Thus, $\sigma \in traces_{t \,||\, \mathcal{A}_\text{i}}$, and the state where σ ends cannot do any observable actions. As t and $\mathcal{A}_\text{i}$ synchronise on every step (except possibly on τ-steps of the implementation), clearly $\sigma \in traces_{\mathcal{A}_\text{i}}$ and $\sigma \in traces_t$. As the final state of σ cannot do any observable actions, it must be a complete trace of t. After all, if this would not be the case, another observation would have been made or another input would have been performed. So, $\sigma \in ctraces_t \cap traces_{\mathcal{A}_\text{i}}$.

($\supseteq$). Let $\sigma \in ctraces_t \cap traces_{\mathcal{A}_\text{i}}$, so $\sigma \in ctraces_t$ and $\sigma \in traces_{\mathcal{A}_\text{i}}$. Then, obviously also $\sigma \in traces_{t \,||\, \mathcal{A}_\text{i}}$. As σ is a complete trace of t, and t and $\mathcal{A}_\text{i}$ must communicate on every action, we have $\sigma \in ctraces_{t \,||\, \mathcal{A}_\text{i}}$. So, by definition $\sigma \in exec_t(\mathcal{A}_\text{i})$. □

Proposition 6.3. *Let $\mathcal{A}_\text{s}$ be a specification, then the annotated test suite $\hat{T} = \{(t, a^{ioco}_{\mathcal{A}_\text{s},t}) \mid t \in \mathcal{T}(\mathcal{A}_\text{s})\}$ is sound for $\mathcal{A}_\text{s}$ with respect to $\sqsubseteq_\text{ioco}$.*

Proof. Let $\mathcal{A}_\text{s}$ be a specification and $\mathcal{A}_\text{i}$ an arbitrary implementation of $\mathcal{A}_\text{s}$. Assuming $v_{\hat{T}}(\mathcal{A}_\text{i}) = fail$, there is an annotated test case $(t, a) \in \hat{T}$ with a trace $\sigma \in exec_t(\mathcal{A}_\text{i})$ such that $\exists \sigma_1 \in traces_{\mathcal{A}_\text{s}}, a! \in L^\delta_\text{O} \; . \; \sigma \sqsupseteq \sigma_1 a! \wedge \sigma_1 a! \notin traces_{\mathcal{A}_\text{s}}$. Since $\sigma \in exec_t(\mathcal{A}_\text{i})$, by Proposition 5.9 also $\sigma \in traces_{\mathcal{A}_\text{i}}$, so $a! \in out_{\mathcal{A}_\text{i}}(\sigma_1)$. However, as $\sigma_1 a! \notin traces_{\mathcal{A}_\text{s}}$, we have $a! \notin out_{\mathcal{A}_\text{s}}(\sigma_1)$. Therefore, $out_{\mathcal{A}_\text{i}}(\sigma_1) \not\subseteq out_{\mathcal{A}_\text{s}}(\sigma_1)$. As $\sigma_1 \in traces_{\mathcal{A}_\text{s}}$, this implies $\mathcal{A}_\text{i} \not\sqsubseteq_\text{ioco} \mathcal{A}_\text{s}$. Thus, $\hat{T}$ is sound for $\mathcal{A}_\text{s}$ with respect to $\sqsubseteq_\text{ioco}$. □

Proposition 6.5. *Let $\hat{T} \subseteq \{(t, a^{ioco}_{\mathcal{A}_\text{s},t}) \mid t \in \mathcal{T}(\mathcal{A}_\text{s})\}$ be a test suite for a specification $\mathcal{A}_\text{s}$, then*

$$\hat{T} \textit{ is complete for } \mathcal{A}_\text{s} \textit{ with respect to } \sqsubseteq_\text{ioco}$$
$$\Leftrightarrow$$
$$\forall \sigma \in canon(traces_{\mathcal{A}_\text{s}}) \; . \; \big(out_{\mathcal{A}_\text{s}}(\sigma) \neq L^\delta_\text{O} \implies \exists (t, a) \in \hat{T} \; . \; \sigma\delta \in t\big)$$

Proof. ($\Leftarrow$) Let $\mathcal{A}_\text{s}$ be a specification and $\hat{T} \subseteq \{(t, a^\text{ioco}_{\mathcal{A}_\text{s},t}) \mid t \in \mathcal{T}(\mathcal{A}_\text{s})\}$ an annotated test suite for $\mathcal{A}_\text{s}$ such that for all $\sigma \in canon(traces_{\mathcal{A}_\text{s}})$ either $out_{\mathcal{A}_\text{s}}(\sigma) = L^\delta_\text{O}$, or there exists an annotated test case $(t, a) \in \hat{T}$ with $\sigma\delta \in t$. To prove that $\hat{T}$ is complete for $\mathcal{A}_\text{s}$ with respect to $\sqsubseteq_\text{ioco}$, we show that $v_{\hat{T}}(\mathcal{A}_\text{i}) = fail$ for every implementation $\mathcal{A}_\text{i} \not\sqsubseteq_\text{ioco} \mathcal{A}_\text{s}$. Let $\mathcal{A}_\text{i}$ be such an implementation, i.e., $\exists \sigma \in traces_{\mathcal{A}_\text{s}} \; . \; out(\mathcal{A}_\text{i}) \not\subseteq out(\mathcal{A}_\text{s})$. Now,

$v_{\hat{T}}(\mathcal{A}_\text{i}) = \textit{fail}$ if and only if there exists an annotated test case $(t, a) \in \hat{T}$ with a trace $\sigma' \in \textit{exec}_t(\mathcal{A}_\text{i})$ such that $a(\sigma') = \textit{fail}$.

By definition, $a^{\text{ioco}}_{\mathcal{A}_\text{s},t}(\sigma') = \textit{fail}$ if and only if it can be written as $\sigma' = \sigma_1 a!\sigma_2$, with $\sigma_1 \in \textit{traces}_{\mathcal{A}_\text{s}}$ and $\sigma_1 a! \notin \textit{traces}_{\mathcal{A}_\text{s}}$. As $\textit{exec}_t(\mathcal{A}_\text{i}) = \textit{ctraces}_t \cap \textit{traces}_{\mathcal{A}_\text{i}}$ (Proposition 5.9), there should be such a trace in both these sets.

As $\mathcal{A}_\text{i}$ was chosen such that $\exists \sigma \in \textit{traces}_{\mathcal{A}_\text{s}}$. $\textit{out}(\mathcal{A}_\text{i}) \not\subseteq \textit{out}(\mathcal{A}_\text{s})$, there is some $a! \in L^\delta_\text{O}$ such that $\sigma a! \in \textit{traces}_{\mathcal{A}_\text{i}}$ and $\sigma a! \notin \textit{traces}_{\mathcal{A}_\text{s}}$. Any trace $\rho \sqsupseteq \sigma a!$ is therefore annotated by *fail*, so if there is a test case $(t, a) \in \hat{T}$ and a trace $\rho \sqsupseteq \sigma a!$ such that $\rho \in \textit{ctraces}_t$ and $\rho \in \textit{traces}_{\mathcal{A}_\text{i}}$, then indeed $v_{\hat{T}}(\mathcal{A}_\text{i}) = \textit{fail}$. If $\sigma a! \in \textit{traces}_t$, there always is at least one continuation $\rho = \sigma a!\sigma_2 \in \textit{ctraces}_t$ (possible with $\sigma_2 = \epsilon$), because of prefix-closedness and the fact that t does not contain infinite traces. Moreover, there always exists such a continuation ρ such that also $\rho \in \textit{traces}_{\mathcal{A}_\text{i}}$, because of the input-enabledness of $\mathcal{A}_\text{i}$ and the fact that test cases always accept all outputs (including quiescence) when observing. So, $\sigma a! \in \textit{traces}_t$ is a sufficient requirement for $v_{\hat{T}}(\mathcal{A}_\text{i}) = \textit{fail}$.

As the erroneous $a!$ can be any output (including δ), for every $a! \in L^\delta_\text{O}$ we need $\sigma a! \in t$. Because of the constraints on test cases, this holds if $\sigma\delta \in t$. Furthermore, σ can be any trace in $\textit{traces}_{\mathcal{A}_\text{s}}$, so in principle $\sigma\delta \in t$ should hold for every $\sigma \in \textit{traces}_{\mathcal{A}_\text{s}}$.

However, we can restrict to observing after every $\sigma \in \textit{canon}(\textit{traces}_{\mathcal{A}_\text{s}})$, because observing two or more consecutive δ actions is never useful. On the one side this follows from the intuition that δ represents the absence of outputs; two times absence is still absence. On the other hand this follows from the definition of QLTSs: $s \xrightarrow{\delta} s'$ implies that $s' \xrightarrow{\delta} s'$, so indeed observing more than one δ never brings a system to any other state than the one arrived in by observing one δ. Moreover, we do not need to observe after traces σ such that $\textit{out}_{\mathcal{A}_\text{s}}(\sigma) = L^\delta_\text{O}$. After all, this implies that all behaviour is considered correct, so there cannot exist an implementation $\mathcal{A}_\text{i}$ such that $\textit{out}(\mathcal{A}_\text{i}) \not\subseteq \textit{out}(\mathcal{A}_\text{s})$.

As all the required observations are in $\hat{T}$, indeed $v_{\hat{T}}(\mathcal{A}_\text{i}) = \textit{fail}$.

($\Rightarrow$) Let $\mathcal{A}_\text{s}$ be a specification and $\hat{T} \subseteq \{(t, a^{\text{ioco}}_{\mathcal{A}_\text{s},t}) \mid t \in \mathcal{T}(\mathcal{A}_\text{s})\}$ an annotated test suite for $\mathcal{A}_\text{s}$ such that there exists a trace $\sigma = a_1 a_2 \dots a_n \in \textit{canon}(\textit{traces}_{\mathcal{A}_\text{s}})$ such that $\textit{out}_{\mathcal{A}_\text{s}}(\sigma) \neq L^\delta_\text{O}$, for which there is no annotated test case $(t, a) \in \hat{T}$ with $\sigma\delta \in t$. We now construct an implementation $\mathcal{A}_\text{i}$ for $\mathcal{A}_\text{s}$ such that $\mathcal{A}_\text{i} \not\sqsubseteq_\text{ioco} \mathcal{A}_\text{s}$, but $v_{\hat{T}}(\mathcal{A}_\text{i}) = \textit{pass}$.

Let $\mathcal{A}_\text{i}$ initially be identical to $\mathcal{A}_\text{s}$ and have k states. We will add states and transitions, in such a way that $\mathcal{A}_\text{i} \not\sqsubseteq_\text{ioco} \mathcal{A}_\text{s}$, but $v_{\hat{T}}(\mathcal{A}_\text{i}) = \textit{pass}$. First we add a state s_{k+1} and change every transition $s \xrightarrow{a_1} s'$ with $s \in S^0$ to $s \xrightarrow{a_1} s_{k+1}$. Then, we add a state s_{k+2} and add a transition $s_{k+1} \xrightarrow{a_2} s_{k+2}$. Moreover, for every existing transition $s \xrightarrow{a} s'$ such that $s \in \textit{reach}_{\mathcal{A}_\text{s}}(a_1)$ and $a \neq a_2$ we add a transition $s_{k+1} \xrightarrow{a} s'$. Then, we add a state s_{k+3} and a transition $s_{k+2} \xrightarrow{a_3} s_{k+3}$. Moreover, for every existing transition $s \xrightarrow{a} s'$ such that $s \in \textit{reach}_{\mathcal{A}_\text{s}}(a_1 a_2)$ and $a \neq a_3$ we add a transition $s_{k+2} \xrightarrow{a} s'$.

We continue like this until we added a state s_{k+n} and the corresponding transitions. Now, let $b! \in L^\delta_\text{O} \setminus \textit{out}_{\mathcal{A}_\text{s}}(\sigma)$ (note that there exists such a $b!$ because of the assumption that $\textit{out}_{\mathcal{A}_\text{s}}(\sigma) \neq L^\delta_\text{O}$). We add a transition $s_{k+n} \xrightarrow{b!} s_{k+n}$. Moreover, for every existing transition $s \xrightarrow{a} s'$ such that $s \in \textit{reach}_{\mathcal{A}_\text{s}}(\sigma)$ and $a \neq b!$ we add a transition $s_{k+n} \xrightarrow{a} s'$.

Note that the restriction to canonical traces was necessary for the above construction to be valid. After all, if two δ actions would occur consecutively, for instance as a_i and a_{i+1}, then it would not be allowed to add the transitions $s_{k+i-1} \xrightarrow{\delta} s_{k+i}$ and $s_{k+i} \xrightarrow{\delta} s_{k+i+1}$; these would violate the restrictions on quiescent transitions.

Now, $\sigma b! \in \mathit{traces}_{\mathcal{A}_\mathrm{i}}$, although $\sigma b! \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, and therefore $\mathit{out}_{\mathcal{A}_\mathrm{i}}(\sigma) \not\subseteq \mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma)$. Thus, as promised $\mathcal{A}_\mathrm{i} \not\sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$. However, as there does not exist an annotated test case $(t, a) \in \hat{T}$ with $\sigma\delta \in t$, this erroneous behaviour will never be observed. As $\mathcal{A}_\mathrm{i}$ does not contain any other erroneous traces, $v_{\hat{T}}(\mathcal{A}_\mathrm{i}) = \mathit{pass}$, so the test suite is incomplete. □

Lemma A.1. *Let $\mathcal{A}_\mathrm{s}$ be a specification, and let $\mathcal{A}'_s = \mathit{det}(\mathcal{A}_\mathrm{s})$. Moreover, let $\mathcal{A}''_s$ be the QLTS obtained from $\mathcal{A}'_s$ by adding to each state s self-loops for all the input actions that are not enabled from s. Then, $\mathcal{A}''_s \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$, and $\mathit{traces}_{\mathcal{A}''_s} \supseteq \mathit{traces}_{\mathcal{A}_\mathrm{s}}$.*

Proof. First of all, note that $\mathcal{A}''_\mathrm{s}$ is input-enabled by definition. To prove that $\mathcal{A}''_\mathrm{s} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$, we show that $\forall \sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}} \,.\, \mathit{out}_{\mathcal{A}''_\mathrm{s}}(\sigma) \subseteq \mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma)$.

By Proposition 4.4 we have $\mathit{traces}_{\mathcal{A}_\mathrm{s}} = \mathit{traces}_{\mathcal{A}'_\mathrm{s}}$, so $\forall \sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}} \,.\, \mathit{out}_{\mathcal{A}'_\mathrm{s}}(\sigma) \subseteq \mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma)$. Let $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ be some trace in the specification. As $\mathcal{A}'_\mathrm{s}$ is deterministic, σ has a unique target state s in $\mathcal{A}'_\mathrm{s}$, so $\mathit{out}_{\mathcal{A}'_\mathrm{s}}(\sigma)$ is given by the outputs enabled in s. As $\mathcal{A}''_\mathrm{s}$ was obtained by adding to every state only transitions labelled by inputs that were not enabled there before, $\mathcal{A}''_\mathrm{s}$ is still deterministic and σ still has s as unique target state. Since only transitions labelled with input actions were added to $\mathcal{A}'_\mathrm{s}$, no additional output actions are enabled in s in $\mathcal{A}''_\mathrm{s}$, so $\mathit{out}_{\mathcal{A}''_\mathrm{s}}(\sigma) = \mathit{out}_{\mathcal{A}'_\mathrm{s}}(\sigma)$, and hence $\mathit{out}_{\mathcal{A}''_\mathrm{s}}(\sigma) \subseteq \mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma)$.

The fact that $\mathit{traces}_{\mathcal{A}''_\mathrm{s}} \supseteq \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ immediately follows from the fact the $\mathit{traces}_{\mathcal{A}_\mathrm{s}} = \mathit{traces}_{\mathcal{A}'_\mathrm{s}}$ and the observation that every trace of $\mathcal{A}'_\mathrm{s}$ is also present in $\mathcal{A}''_\mathrm{s}$ (as the latter is obtained by *adding* transitions to the former). □

Proposition 6.6. *Let $\mathcal{A}_\mathrm{s}$ be a specification, then the annotated test suite $\hat{T} = \{(t, a^{ioco}_{\mathcal{A}_\mathrm{s},t}) \mid t \in \mathcal{T}(\mathcal{A}_\mathrm{s})\}$ is consistent for $\mathcal{A}_\mathrm{s}$ with respect to $\sqsubseteq_\mathrm{ioco}$.*

Proof. Let $\mathcal{A}_\mathrm{s}$ be a specification and $\hat{T} = \{(t, a^\mathrm{ioco}_{\mathcal{A}_\mathrm{s},t}) \mid t \in \mathcal{T}(\mathcal{A}_\mathrm{s})\}$ the maximal annotated test suite of which every test case t has been annotated according to $a^\mathrm{ioco}_{\mathcal{A}_\mathrm{s},t}$. Let $\hat{t} = (t, a)$ be any test case of $\hat{T}$, and let $\sigma \in \mathit{ctraces}_t$ be any complete trace of t such that $a(\sigma) = \mathit{pass}$. For consistency it must then hold that there exists an implementation $\mathcal{A}_\mathrm{i}$ such that $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{i}}$ and $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$. We will construct such an implementation.

Initially, we will take $\mathcal{A}_\mathrm{i} = \mathcal{A}''_\mathrm{s}$, where $\mathcal{A}''_\mathrm{s}$ is obtained from $\mathcal{A}_\mathrm{s}$ as described in Lemma A.1. By this lemma we know that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$ and $\mathit{traces}_{\mathcal{A}_\mathrm{i}} \supseteq \mathit{traces}_{\mathcal{A}_\mathrm{s}}$. If $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ then also $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{i}}$ and we are done, so assume that $\sigma \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$.

By definition of $a^\mathrm{ioco}_{\mathcal{A}_\mathrm{s},t}$ it follows from $a(\sigma) = \mathit{pass}$ that there does not exist a trace $\sigma_1 \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ and action $a! \in L^\delta_\mathrm{O}$ such that $\sigma \sqsupseteq \sigma_1 a! \wedge \sigma_1 a! \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$. This implies that either $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ (since in that case obviously we cannot find a prefix that is not in $\mathit{traces}_{\mathcal{A}_\mathrm{s}}$), or there exists an $a? \in L_\mathrm{I}$, $\sigma' \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, and $\sigma'' \in L^*$ such that $\sigma = \sigma' a? \sigma''$ and $\sigma' a? \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$. Given that we assumed $\sigma \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, the latter must be the case.

Now, as $\mathit{traces}_{\mathcal{A}_\mathrm{i}} \supseteq \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, clearly $\sigma' \in \mathit{traces}_{\mathcal{A}_\mathrm{i}}$. As $\sigma' a? \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ and $\mathcal{A}_\mathrm{i}$ was obtained by adding self-loops to the determinisation of $\mathcal{A}_\mathrm{s}$, there must be a self-loop labelled $a?$ in $\mathcal{A}_\mathrm{i}$ from the (unique) target state s of σ'. We will now remove this single self-loop from $\mathcal{A}_\mathrm{i}$ and add a new state s' to $\mathcal{A}_\mathrm{i}$, as well as a transition labelled $a?$ from s to s'. Then, we add new states and transitions from s' mimicking the trace σ''.

Clearly, after this transformation we have $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{i}}$. Moreover, still $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$, as we only changed the behaviour after the trace $\sigma' a?$, which is not in $\mathit{traces}_{\mathcal{A}_\mathrm{s}}$. Therefore, as $\sqsubseteq_\mathrm{ioco}$ only requires $\mathit{out}_{\mathcal{A}_\mathrm{i}}(\sigma) \subseteq \mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma)$ for every $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, this transformation cannot make $\mathcal{A}_\mathrm{i}$ not ioco-implement $\mathcal{A}_\mathrm{s}$ anymore. □

Theorem 6.8. *Let $\mathcal{A}_\mathrm{s}$ be a specification and $\mathcal{A}_\mathrm{i}$ an implementation of $\mathcal{A}_\mathrm{s}$. Then, $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$ if and only if for every trace $\sigma \in traces_{\mathcal{A}_\mathrm{i}}$ it holds that*

$$\sigma \notin traces_{\mathcal{A}_\mathrm{s}} \implies \exists \sigma' \in traces_{\mathcal{A}_\mathrm{s}}, a? \in L_\mathrm{I} \;.\; \sigma' a? \sqsubseteq \sigma \wedge \sigma' a? \notin traces_{\mathcal{A}_\mathrm{s}}$$

Proof. ($\Leftarrow$) Let $\mathcal{A}_\mathrm{s}$ be a specification and $\mathcal{A}_\mathrm{i}$ an implementation of $\mathcal{A}_\mathrm{s}$ such that the above condition holds. We prove that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$, i.e., that for every $\sigma \in traces_{\mathcal{A}_\mathrm{s}}$ it holds that $out_{\mathcal{A}_\mathrm{i}}(\sigma) \subseteq out_{\mathcal{A}_\mathrm{s}}(\sigma)$. Let $\sigma \in traces_{\mathcal{A}_\mathrm{s}}$ be a trace, then either $out_{\mathcal{A}_\mathrm{i}}(\sigma) = \emptyset$ or $out_{\mathcal{A}_\mathrm{i}}(\sigma) \neq \emptyset$. In the first case, trivially $out_{\mathcal{A}_\mathrm{i}}(\sigma) \subseteq out_{\mathcal{A}_\mathrm{s}}(\sigma)$. In the second case, let $b! \in out_{\mathcal{A}_\mathrm{i}}(\sigma)$. Then it must hold that $\sigma b! \in traces_{\mathcal{A}_\mathrm{i}}$, so because of our assumption either $\sigma b! \in traces_{\mathcal{A}_\mathrm{s}}$, or $\sigma b! = \sigma' a? \sigma''$ such that $\sigma' \in traces_{\mathcal{A}_\mathrm{s}}$ and $\sigma' a? \notin traces_{\mathcal{A}_\mathrm{s}}$. It is easy to see that the latter cannot be true because $\sigma \in traces_{\mathcal{A}_\mathrm{s}}$ and $b! \notin L_\mathrm{I}$. So, $\sigma b! \in traces_{\mathcal{A}_\mathrm{s}}$, and therefore $b! \in out_{\mathcal{A}_\mathrm{s}}(\sigma)$.

($\Rightarrow$) Let $\mathcal{A}_\mathrm{s}$ be a specification and $\mathcal{A}_\mathrm{i}$ an implementation of $\mathcal{A}_\mathrm{s}$ such that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$. By definition $out_{\mathcal{A}_\mathrm{i}}(\sigma) \subseteq out_{\mathcal{A}_\mathrm{s}}(\sigma)$ for all $\sigma \in traces_{\mathcal{A}_\mathrm{s}}$. We now prove that for any $\sigma \in traces_{\mathcal{A}_\mathrm{i}}$ it holds that $\sigma \notin traces_{\mathcal{A}_\mathrm{s}}$ implies that $\sigma = \sigma' a? \sigma''$ such that $a? \in L_\mathrm{I} \wedge \sigma' \in traces_{\mathcal{A}_\mathrm{s}} \wedge \sigma' a? \notin traces_{\mathcal{A}_\mathrm{s}}$, using induction on the length of σ.

Base case ($|\sigma| = 0$): When $|\sigma| = 0$ it follows that $\sigma = \epsilon$, which is obviously a trace of every specification, so $\sigma \in traces_{\mathcal{A}_\mathrm{s}}$.

Induction hypothesis: For every $\sigma \in traces_{\mathcal{A}_\mathrm{i}}$ such that $|\sigma| = k$ it holds that $\sigma \notin traces_{\mathcal{A}_\mathrm{s}}$ implies that $\sigma = \sigma' a? \sigma''$ such that $a? \in L_\mathrm{I} \wedge \sigma' \in traces_{\mathcal{A}_\mathrm{s}} \wedge \sigma' a? \notin traces_{\mathcal{A}_\mathrm{s}}$.

Inductive case ($|\sigma| = k + 1$): Let $\sigma \in traces_{\mathcal{A}_\mathrm{i}}$ such that $|\sigma| = k+1$ and $\sigma \notin traces_{\mathcal{A}_\mathrm{s}}$. We prove that $\sigma = \sigma' a? \sigma''$ such that $a? \in L_\mathrm{I} \wedge \sigma' \in traces_{\mathcal{A}_\mathrm{s}} \wedge \sigma' a? \notin traces_{\mathcal{A}_\mathrm{s}}$. Clearly, we can write $\sigma = \sigma_k a$, where $|\sigma_k| = k$ and $a \in L$. By the induction hypothesis $\sigma_k \in traces_{\mathcal{A}_\mathrm{s}} \vee \sigma_k = \sigma' b? \sigma''$ such that $b? \in L_\mathrm{I} \wedge \sigma' \in traces_{\mathcal{A}_\mathrm{s}} \wedge \sigma' b? \notin traces_{\mathcal{A}_\mathrm{s}}$. In case the second disjunct is true, then $\sigma = \sigma' b? \sigma'' a$ and we are done. So, from now on we assume that $\sigma_k \in traces_{\mathcal{A}_\mathrm{s}}$. Because of the definition of $\sqsubseteq_\mathrm{ioco}$ and the assumption that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$ we then know that $out_{\mathcal{A}_\mathrm{i}}(\sigma_k) \subseteq out_{\mathcal{A}_\mathrm{s}}(\sigma_k)$. Combining this with the fact that $\sigma_k a \in traces_{\mathcal{A}_\mathrm{i}}$ and $\sigma_k a \notin traces_{\mathcal{A}_\mathrm{s}}$, it must hold that $a \in L_\mathrm{I}$. Now, indeed $\sigma = \sigma_k a \epsilon$ with $a \in L_\mathrm{I} \wedge \sigma_k \in traces_{\mathcal{A}_\mathrm{s}} \wedge \sigma_k a \notin traces_{\mathcal{A}_\mathrm{s}}$. □

Corollary 6.9. *Let $\mathcal{A}_\mathrm{s}$ be an input-enabled specification and $\mathcal{A}_\mathrm{i}$ an implementation of $\mathcal{A}_\mathrm{s}$. Then, $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s} \Leftrightarrow traces_{\mathcal{A}_\mathrm{i}} \subseteq traces_{\mathcal{A}_\mathrm{s}}$.*

Proof. Let $\mathcal{A}_\mathrm{s}$ be an input-enabled specification and $\mathcal{A}_\mathrm{i}$ an implementation of $\mathcal{A}_\mathrm{s}$. Theorem 6.8 showed that $\mathcal{A}_\mathrm{i} \sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$ if and only if for every $\sigma \in traces_{\mathcal{A}_\mathrm{i}}$ it holds that

$$\sigma \in traces_{\mathcal{A}_\mathrm{s}} \vee$$
$$\sigma = \sigma' a? \sigma'' \text{ such that } a? \in L_\mathrm{I} \wedge \sigma' \in traces_{\mathcal{A}_\mathrm{s}} \wedge \sigma' a? \notin traces_{\mathcal{A}_\mathrm{s}}$$

However, because $\mathcal{A}_\mathrm{s}$ is input-enabled, the second disjunct can never be fulfilled by any trace. After all, when $\sigma' \in traces_{\mathcal{A}_\mathrm{s}}$, then also $\sigma' a? \in traces_{\mathcal{A}_\mathrm{s}}$ for every $a? \in L_\mathrm{I}$. Therefore, the result simplifies to that for every $\sigma \in traces_{\mathcal{A}_\mathrm{i}}$ it holds that $\sigma \in traces_{\mathcal{A}_\mathrm{s}}$, that is, $traces_{\mathcal{A}_\mathrm{i}} \subseteq traces_{\mathcal{A}_\mathrm{s}}$. □

Proposition 6.11. *Let $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ be QLTSs such that $\mathcal{A}$ and $\mathcal{B}$ are input-enabled, then*

$$\mathcal{A} \sqsubseteq_{\text{ioco}} \mathcal{B} \wedge \mathcal{B} \sqsubseteq_{\text{ioco}} \mathcal{C} \Rightarrow \mathcal{A} \sqsubseteq_{\text{ioco}} \mathcal{C}$$

Proof. Let $\mathcal{A}, \mathcal{B}$ and $\mathcal{C}$ be QLTSs such that $\mathcal{A}$ and $\mathcal{B}$ are input-enabled, $\mathcal{A} \sqsubseteq_{\text{ioco}} \mathcal{B}$ and $\mathcal{B} \sqsubseteq_{\text{ioco}} \mathcal{C}$. Then, by Theorem 6.8 and Corollary 6.9 we know that $\mathit{traces}_{\mathcal{A}} \subseteq \mathit{traces}_{\mathcal{B}}$, and that for all $\sigma \in \mathit{traces}_{\mathcal{B}}$ it holds that $\sigma \notin \mathit{traces}_{\mathcal{C}} \implies \sigma = \sigma' a? \sigma''$ such that $a? \in L_{\text{I}} \wedge \sigma' \in \mathit{traces}_{\mathcal{C}} \wedge \sigma' a? \notin \mathit{traces}_{\mathcal{C}}$.

Now, let $\sigma \in \mathit{traces}_{\mathcal{A}}$. We now prove that $\sigma \notin \mathit{traces}_{\mathcal{C}} \implies \sigma = \sigma' a? \sigma''$ such that $a? \in L_{\text{I}} \wedge \sigma' \in \mathit{traces}_{\mathcal{C}} \wedge \sigma' a? \notin \mathit{traces}_{\mathcal{C}}$. Assume that $\sigma \notin \mathit{traces}_{\mathcal{C}}$.

From $\mathit{traces}_{\mathcal{A}} \subseteq \mathit{traces}_{\mathcal{B}}$ it follows that $\sigma \in \mathit{traces}_{\mathcal{B}}$, and from $\sigma \notin \mathit{traces}_{\mathcal{C}}$ and the earlier observation that for all $\sigma \in \mathit{traces}_{\mathcal{B}}$ it holds that $\sigma \notin \mathit{traces}_{\mathcal{C}} \implies \sigma = \sigma' a? \sigma''$ such that $a? \in L_{\text{I}} \wedge \sigma' \in \mathit{traces}_{\mathcal{C}} \wedge \sigma' a? \notin \mathit{traces}_{\mathcal{C}}$ it immediately follows that $\sigma = \sigma' a? \sigma''$ such that $a? \in L_{\text{I}} \wedge \sigma' \in \mathit{traces}_{\mathcal{C}} \wedge \sigma' a? \notin \mathit{traces}_{\mathcal{C}}$. □

Theorem 7.3. *Let $\mathcal{A}_{\text{s}}$ be a specification, and $t = \text{batchGen}(\mathcal{A}_{\text{s}}, \epsilon)$. Then, t is a fail-fast and input-minimal test case for $\mathcal{A}_{\text{s}}$.*

Proof. Let $\mathcal{A}_{\text{s}}$ be a specification, and $t = \text{batchGen}(\mathcal{A}_{\text{s}}, \epsilon)$. First, we prove that t indeed is a test case for $\mathcal{A}_{\text{s}}$. For this to hold (1) it should be a prefix-closed subset of L^*, (2) it should not contain an infinite increasing sequence $\sigma_0 \sqsubset \sigma_1 \sqsubset \sigma_2 \sqsubset \dots$, and (3) it should be such that for all $\sigma \in t$, either

1. $\{a \in L \mid \sigma a \in t\} = \emptyset$; or
2. $\{a \in L \mid \sigma a \in t\} = L_{\text{O}}^{\delta}$; or
3. $\{a \in L \mid \sigma a \in t\} = \{a?\} \cup L_{\text{O}}$ for some $a? \in L_{\text{I}}$.

(1) Let $\sigma \in t$ such that $\sigma = \sigma' a$, with $a \in L$. We show that also $\sigma' \in t$, and it follows by induction that t is prefix-closed.
If $\sigma' = \epsilon$, then clearly it is in t. After all, no matter which one of the three nondeterministic choices is chosen during the first iteration, ϵ is always added to t (either by line 2, 4 or 11).
If $|\sigma' a| = k > 1$, then $\sigma' a$ can only have been added to t by means of k recursive calls (line 7, line 11, or line 14) followed by line 2, or by $k - 1$ recursive calls followed by either line 8 or line 15. As during each recursive call also ϵ is returned (because of either line 2, 4 or 11), this results in not only $\sigma' a$ but also σ'.

(2) As every iteration of the algorithm increases the depth of the test case by one, a test case t obtained by running the algorithm (a finite amount of time) can never have an infinite increasing sequence.

(3) If $\sigma \in t$, then longer traces σa arrive in t by means of the recursive calls (line 7, line 11, and line 14). Such a recursive call appends traces σ' to an action $a?$ or $b!$ that result from using the algorithm. Looking at the three nondeterministic choices, we see that this set of traces is either $\{\epsilon\}$ (so $\{a \in L \mid \sigma a \in t\} = \emptyset$), or it consists of traces $b!\sigma''$ for all $b! \in L_{\text{O}}^{\delta}$ (so $\{a \in L \mid \sigma a \in t\} = L_{\text{O}}^{\delta}$), or it consists of traces $b!\sigma''$ for all $b! \in L_{\text{O}}$ and traces $a?\sigma''$ for some $a? \in L_{\text{I}}$ (so $\{a \in L \mid \sigma a \in t\} = \{a?\} \cup L_{\text{O}}$). This exactly corresponds to the requirements for t to be a test case.

That t is fail-fast follows directly from the if-statements on line 6 and line 13. After all, they exactly make sure that recursive calls to extend a trace $\sigma b!$ are only made in case $\sigma b! \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$.

That t is input-minimal follows directly from the guard of the third nondeterministic choice; because of this guard, no trace $\sigma a?$ with $\sigma a? \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ will ever be added to the test case. $\square$

Theorem 7.4. *Let $\mathcal{A}_\mathrm{s}$ be a specification, and T the set of all linear test cases that can be generated using Algorithm 1. Then, the annotated test suite $\hat{T} = \{(t, a^{ioco}_{\mathcal{A}_\mathrm{s},t}) \mid t \in T\}$ is complete for $\mathcal{A}_\mathrm{s}$ with respect to $\sqsubseteq_\mathrm{ioco}$.*

Proof. Let $\mathcal{A}_\mathrm{s}$ be a specification, and T the set of all linear test cases that can be generated using Algorithm 1. By Proposition 6.5 we know that $\hat{T}$ is complete for $\mathcal{A}_\mathrm{s}$ with respect to $\sqsubseteq_\mathrm{ioco}$ if for all $\sigma \in \mathit{canon}(\mathit{traces}_{\mathcal{A}_\mathrm{s}})$ such that $\mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma) \neq L^\delta_\mathrm{O}$ there exists an annotated test case $(t, a) \in \hat{T}$ such that $\sigma\delta \in t$.

Let $\sigma = a_1 a_2 \ldots a_n \in \mathit{canon}(\mathit{traces}_{\mathcal{A}_\mathrm{s}})$. We now show that indeed there exists a linear test case $t \in T$ such that $\sigma\delta \in t$ by constructing this test case. We will construct it in such a way that σ will be the main trace of t.

In the first iteration, we resolve the nondeterminism based on the action a_1. If $a_1 \in L_\mathrm{I}$, then we choose to stimulate a_1. This results in several recursive calls; one for the history a_1 and one for every $b! \in L_\mathrm{O}$. For all the outputs $b!$ the next choice should be to return ϵ; that way, t remains linear as all traces only deviate one action from the main trace σ. If $a_1 \in L_\mathrm{O}$, then we choose to observe. This results again in several recursive calls; one for every $b! \in L^\delta_\mathrm{O}$. Now, for all outputs $b! \neq a_1$ the recursive call should return ϵ for t to remain linear.

In the second iteration, caused by the recursive call with history a_1, the same strategy should be applied. Finally, at the $(n+1)^\mathrm{th}$ iteration, having history σ, choose to observe. This causes $\sigma\delta$ to be added to t. Now return ϵ in all remaining recursive calls to terminate the algorithm.

Note that indeed t is linear with main trace σ, as for every deviation from it we immediately return ϵ. $\square$

Theorem 8.1. *Algorithm 2 is sound with respect to $\sqsubseteq_\mathrm{ioco}$.*

Proof. Note that the variable σ keeps track of the trace exhibited by the implementation thus far. The only way for the algorithm to return *fail* is when $\sigma \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$ after an observation. Note that in this case we can always write $\sigma = \sigma' b!$. It is easy to see that up to the point of returning *fail* only inputs were provided and correct outputs were observed, otherwise a *fail* would have been returned earlier. As only correct inputs are provided and the algorithm terminates as soon as an unexpected output is observed, it follows that $\sigma \in \mathit{traces}_{\mathcal{A}_\mathrm{s}}$. As we have observed that $b! \in \mathit{out}_{\mathcal{A}_\mathrm{i}}(\sigma)$, and as we know that $b! \notin \mathit{out}_{\mathcal{A}_\mathrm{s}}(\sigma)$ because $\sigma b! \notin \mathit{traces}_{\mathcal{A}_\mathrm{s}}$, by definition $\mathcal{A}_\mathrm{i} \not\sqsubseteq_\mathrm{ioco} \mathcal{A}_\mathrm{s}$. $\square$