Testing Divergent Transition Systems[⋆]

Ed Brinksma, Mariëlle I. A. Stoelinga, and Mark Timmer

Formal Methods and Tools, Faculty of EEMCS
University of Twente, The Netherlands
{h.brinksma, marielle, m.timmer}@utwente.nl

Abstract. We revisit model-based testing for labelled transition systems in the context of specifications that may contain divergent behaviour, i.e., infinite paths of internal computations. The standard approach based on the theory of input-output conformance, known as the ioco-framework, cannot deal with divergences directly, as it restricts specifications to strongly convergent transition systems. Using the model of Quiescent Input Output Transition Systems (QIOTSs), we can handle divergence successfully in the context of quiescence. Quiescence is a fundamental notion that represents the situation that a system is not capable of producing any output, if no prior input is provided, representing lack of productive progress. The correct treatment of this situation is the cornerstone of the success of testing in the context of systems that are input-enabled, i.e., systems that accept all input actions in any state. Our revised treatment of quiescence also allows it to be preserved under determinization of a QIOTS. This last feature allows us to reformulate the standard ioco-based testing theory and algorithms in terms of classical trace-based automata theory, including finite state divergent computations.

1 Introduction

Quiescence is a fundamental notion that represents the situation that a system is not capable of producing any output, if no prior input is provided, representing lack of productive progress. The correct treatment of this situation is the cornerstone of the success of testing in the context of systems that are input-enabled, i.e., systems that accept all input actions in any state. The standard approach to model-based testing of labelled transition systems, based on the theory of input-output conformance, known as the ioco-framework, is based on the explicit treatment of quiescence as an observable property of a system under test, by treating inaction with respect to output as a special kind of null action.

The proper treatment of quiescence is complicated by the phenomenon of divergence. Transition systems are said to be divergent if their computation traces include infinite sequences of internal steps, i.e., steps that are not observable

[⋆] This research has been partially funded by NWO under grants 612.063.817 (SYRUP), Dn 63-257 (ROCKS) and 12238 (ArRangeer), and by the EU under grant 318490 (SENSATION).

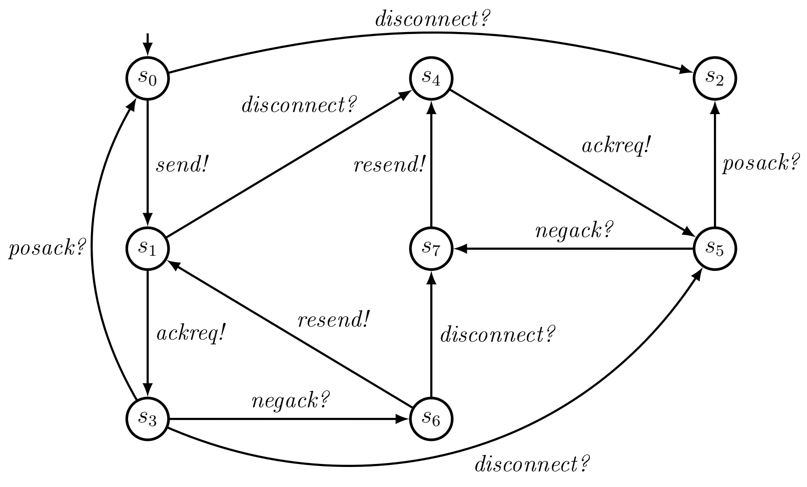


Fig. 1. A simple communication system with acknowledgements.

as part of the communication of the system with its environment. The possibility of such infinite internal computations can be a source of quiescence. The `ioco`-framework cannot deal with divergent behaviour in specifications directly: it requires specifications to be strongly convergent, i.e., they must contain only finite sequences of internal actions in computations [1]. Divergence, however, does often occur in practice.

Example 1.1. Consider for instance the simple communication system with acknowledgements and retransmissions shown in Figure 1. When hiding all actions related to retransmissions (*ackreq, negack, posack*, and *resend*) by renaming them to the internal action τ, we obtain a specification in which infinite cycles of internal actions are possible in the two 'triangles' of the system, cycling through s_1, s_3, s_6, and s_4, s_5, s_7, respectively. □

We use this example to show that there are differences between divergences that matter in the context of testing. It would be too simple, for example, to create observations of quiescence for every τ-loop. In the example, the infinite execution of the loops of the triangles can be considered unfair, in the sense that they would ignore the transitions of the (hidden) *posack* action. Our approach will be that such unfair divergences cannot occur, expecting internal or output actions to never be delayed infinitely many times. This means that the triangle s_1, s_3, s_6 will ultimately enable a *send* action in state s_0, and will not cause quiescence, whereas the control cycling through s_4, s_5, s_7 will eventually reach s_2, which is quiescent. Of course, some τ-loops can occur fairly. If we turn the above example into an input-enabled system, which will be the typical case in the context of this theory, then self-loops with missing input actions will be added to states. In particular, state s_2 will get three extra transitions to itself, labelled with *posack?, negack?* and *disconnect?*. Upon hiding, the first two become τ-loops, and cycling continuously through both of them is a fair execution of divergence, as this does not compete with other locally controlled actions such as other outgoing τ-actions or outputs. The theory that we wish to develop must deal with this variety in divergences and all the subtleties involved.

A way forward has been proposed in [2] in the form of the model of Divergent Quiescent Transition Systems (DQTSs). This model is formulated using input-output automata (IOA), introduced by Lynch et al. [3], and improves the existing theory in three respects. Firstly, it removes the restriction to strongly convergent specifications; secondly, it deals correctly with the notion of quiescence in the presence of divergence, i.e., it distinguishes between infinite computations that can block output from occurring (and therefore should signal quiescence) vs. those that do not; and thirdly, it revises the definition of quiescence so that it is preserved under determinization, allowing the reformulation of `ioco`-based testing theory including divergence and the related test generation algorithms in terms of classical trace-based automata theory.

Our purpose in this paper is also threefold. First, we want to obtain a simplified version of the model that does not need the full works of the IOA framework, such as the possibility to name internal actions and to specify fairness constraints in great generality using task partitions. Instead, and this is our second goal, when dealing with divergences it should use the notion of fair execution that is implicitly behind most labelled transition system modelling involving just a single anonymous internal step τ, as for example captured by Milner's weak bisimulation equivalence [4]. Finally, we want to connect this theory to the standard `ioco`-algorithms for test generation [5], which is only suggested by the work in [2], but has not yet been carried out. In doing so, this paper is nicely representative of the work on model-based testing at the University of Twente during the long period of collaboration that we have had with Kim Larsen.

Origins. The notion of quiescence was first introduced by Vaandrager [6] to obtain a natural extension of blocking states: if a system is input-enabled (i.e., always ready to receive inputs), then no states are blocking, since each state has outgoing input transitions. Quiescence models the fact that a state is blocking when considering only the internal and output actions. In the context of model-based testing, Tretmans introduced *repetitive quiescence* [7, 1]. This notion emerged from the need to continue testing, even in a quiescent state. To accommodate this, Tretmans introduced the *Suspension Automaton* (SA) as an auxiliary concept [8]. An SA is obtained from an Input-Output Transition System (IOTS) by first adding to each quiescent state a self-loop labelled by the quiescence label δ and by subsequently determinizing the model. As stated above, SAs cannot cope with divergence, since divergence leads to new quiescent states. In an attempt to remedy this situation, the TGV framework [9] handles divergence by adding δ-labelled self-loops to such states. However, this treatment is not satisfactory in our opinion: quiescence due to divergence can in [9] be followed by an output action, which we find counterintuitive.

Overview of the paper. The rest of the paper is organized as follows. Section 2 introduces the QIOTS model, and Section 3 provides operations and properties for them. Sections 4, 5 and 6 present the notions of test cases for QIOTSs, an overview of testing with respect to `ioco` and algorithms for generating test cases, respectively. Conclusions and future work are presented in Section 7.

This paper simplifies and unites the concepts from [2] and [5] (partly by the same authors); parts of this paper overlap with these works.

2 Quiescent Input Output Transition Systems

Preliminaries Given a set L, we use L^* to denote the set of all *finite sequences* $\sigma = a_1\, a_2\, \ldots\, a_n$ over L. We write $|\sigma| = n$ for the *length* of σ, and ϵ for the *empty sequence*, and let $L^+ = L^* \setminus \{\,\epsilon\,\}$. We let L^ω denote the set of all *infinite sequences* over L, and use $L^\infty = L^* \cup L^\omega$. Given two sequences $\rho \in L^*$ and $\upsilon \in L^\infty$, we denote the *concatenation* of ρ and υ by $\rho\,\upsilon$. The *projection of an element* $a \in L$ *on* $L' \subseteq L$, denoted $a \upharpoonright L'$, is a if $a \in L'$ and ϵ otherwise. The projection of a sequence $\sigma = a\,\sigma'$ is defined inductively by $(a\,\sigma') \upharpoonright L' = (a \upharpoonright L')(\sigma' \upharpoonright L')$, and the projection of a set of sequences Z is defined as the set of projections.

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 use $\wp(L)$ to denote the *power set* of L. Finally, we use the notation $\exists^\infty$ for 'there exist infinitely many'.

2.1 Basic model and definitions

Quiescent Input Output Transition Systems (QIOTSs) are labelled transition systems that model quiescence, i.e., the absence of outputs or internal transitions, via a special δ-action. Internal actions, in turn, are all represented by the special τ-action. Thus, QIOTSs are a variety of Input Output Transition Systems of the `ioco`-framework, and an adaptation of the DQTS model of [2], which, in turn, is based on the well-known model of Input-Output Automata [10, 3].

Definition 2.1 (Quiescent Input Output Transition Systems). *A* Quiescent Input Output Transition System *(QIOTS) is a tuple* $\mathcal{A} = \langle\, S, s^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$*, where* S *is a set of states;* $s^0 \in S$ *is its initial state;* L^{I} *and* L^{O} *are disjoint sets of input and output labels, respectively; and* $\rightarrow \subseteq S \times L \times S$ *is the transition relation, where* $L = L^{\mathrm{I}} \cup L^{\mathrm{O}} \cup \{\,\tau, \delta\,\}$*. We assume* $\delta, \tau \notin (L^{\mathrm{I}} \cup L^{\mathrm{O}})$ *and* $\delta \neq \tau$*, and use* $L^{\mathrm{O}}_\delta = L^{\mathrm{O}} \cup \{\,\delta\,\}$*. We refer to* $(L^{\mathrm{I}}, L^{\mathrm{O}}_\delta)$ *as the* action signature *of* $\mathcal{A}$*.*

The following two requirements apply:

1. *A QIOTS must be* input-enabled*, i.e., for each* $s \in S$ *and* $a \in L^{\mathrm{I}}$*, there exists an* $s' \in S$ *such that* $(s, a, s') \in \rightarrow$*.*
2. *A QIOTS must be well-formed. Well-formedness requires technical preparation and is defined in Section 2.3.*

We write $\mathcal{QIOTS}(L^{\mathrm{I}}, L^{\mathrm{O}}_\delta)$ *for the set of all possible QIOTSs over the action signature* $(L^{\mathrm{I}}, L^{\mathrm{O}}_\delta)$*.*

Semantically, QIOTSs assume progress. That is, QIOTSs are not allowed to remain idle forever when output or internal actions are enabled. Without this assumption, each state would be potentially quiescent. All sets in the definition of QIOTSs can potentially be uncountable.

Given a QIOTS $\mathcal{A}$, we denote its components by $S_{\mathcal{A}}, s^0_{\mathcal{A}}, L^{\mathrm{I}}_{\mathcal{A}}, L^{\mathrm{O}}_{\mathcal{A}}, \rightarrow_{\mathcal{A}}$. We omit the subscript when it is clear from the context.

Actions. We use the terms label and action interchangeably. We often suffix a question mark (?) to input labels and an exclamation mark (!) to output labels. These are, however, not part of the label. The label τ represents an internal action. Output and internal actions are called *locally controlled*, because their occurrence is under the control of the QIOTS. The special label δ is used to denote the occurrence of quiescence.

We use the standard notations for transitions.

Definition 2.2 (Transitional notations). *Let $\mathcal{A}$ be a QIOTS with $s, s' \in S$, $a, a_i \in L$, $b, b_i \in L \setminus \{\tau\}$, and $\sigma \in (L \setminus \{\tau\})^+$, then:*

$$\begin{array}{lll}
s \xrightarrow{a} s' & =_{\mathrm{def}} & (s, a, s') \in \rightarrow \\
s \xrightarrow{a} & =_{\mathrm{def}} & \exists\, s'' \in S \;.\; s \xrightarrow{a} s'' \\
s \not\xrightarrow{a} & =_{\mathrm{def}} & \nexists\, s'' \in S \;.\; s \xrightarrow{a} s'' \\
s \xrightarrow{a_1 \cdot \ldots \cdot a_n} s' & =_{\mathrm{def}} & \exists\, s_0, \ldots, s_n \in S \;.\; s = s_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} s_n = s' \\
s \xRightarrow{\epsilon} s' & =_{\mathrm{def}} & s = s' \text{ or } s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \xRightarrow{b} s' & =_{\mathrm{def}} & \exists\, s_0, s_1 \in S \;.\; s \xRightarrow{\epsilon} s_0 \xrightarrow{b} s_1 \xRightarrow{\epsilon} s' \\
s \xRightarrow{b_1 \cdot \ldots \cdot b_n} s' & =_{\mathrm{def}} & \exists\, s_0, \ldots, s_n \in S \;.\; s = s_0 \xRightarrow{b_1} \cdots \xRightarrow{b_n} s_n = s'
\end{array}$$

If $s \xrightarrow{a}$, we say that a is enabled *in s. We use $L(s)$ to denote the set of all actions $a \in L$ that are enabled in state $s \in S$, i.e., $L(s) = \{ a \in L \mid s \xrightarrow{a} \}$. The notions are lifted to infinite traces in the obvious way.*

We use the following language notations for QIOTSs and their behaviour.

Definition 2.3 (Language notations). *Let $\mathcal{A}$ be a QIOTS, then:*

- *A finite path in $\mathcal{A}$ is a sequence $\pi = s_0\, a_1\, s_1\, a_2\, s_2\, \ldots\, s_n$ such that $s_{i-1} \xrightarrow{a_i} s_i$ for all $1 \leq i \leq n$. Infinite paths are defined analogously. The set of all paths in $\mathcal{A}$ is denoted paths$(\mathcal{A})$.*
- *Given any path, we write first$(\pi) = s_0$. Also, we denote by states(π) the set of states that occur on π, and by ω-states(π) the set of states that occur infinitely often. That is, ω-states$(\pi) = \{ s \in$ states$(\pi) \mid \exists^\infty j \;.\; s_j = s \}$.*
- *We define trace$(\pi) = \pi \restriction (L \setminus \{\tau\})$, and say that trace$(\pi)$ is the* trace *of π. For every $s \in S$, traces(s) is the set of all traces corresponding to paths that start in s, i.e., traces$(s) = \{$ trace$(\pi) \mid \pi \in$ paths$(\mathcal{A}) \wedge$ first$(\pi) = s \}$. We define traces$(\mathcal{A})$ = traces(s^0), and say that two QIOTSs $\mathcal{B}$ and $\mathcal{C}$ are* trace-equivalent*, denoted $\mathcal{B} \approx_{\mathrm{tr}} \mathcal{C}$, if traces$(\mathcal{B})$ = traces$(\mathcal{C})$.*
- *For a finite trace σ and state $s \in S$, reach(s, σ) denotes the set of states in $\mathcal{A}$ that can be reached from s via σ, i.e., reach$(s, \sigma) = \{ s' \in S \mid s \xRightarrow{\sigma} s' \}$. For a set of states $S' \subseteq S$, we define reach$(S', \sigma) = \bigcup_{s \in S'}$ reach(s, σ).*

When needed, we add subscripts to indicate the QIOTS these notions refer to.

Definition 2.4 (Determinism). *A QIOTS $\mathcal{A}$ is* deterministic *if $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ imply $a \neq \tau$ and $s' = s''$, for all $s, s', s'' \in S$ and $a \in L$. Otherwise, $\mathcal{A}$ is* nondeterministic.

Each QIOTS has an obviously trace-equivalent deterministic QIOTS. Determinization is carried out using the well-known subset construction procedure. This construction yields a system in which every state has a unique target state per action, and internal transitions are not present anymore.

Definition 2.5 (Determinization). *The* determinization *of a QIOTS $\mathcal{A} = \langle S, s^0, L^\mathrm{I}, L^\mathrm{O}, \rightarrow \rangle$ is the QIOTS $det(\mathcal{A}) = \langle \wp(S)^+, \{ s^0 \}, L^\mathrm{I}, L^\mathrm{O}, \rightarrow_\mathrm{D} \rangle$, with $\wp(S)^+ = \wp(S) \setminus \varnothing$ and $\rightarrow_\mathrm{D} = \{ (U, a, V) \in \wp(S)^+ \times L \times \wp(S)^+ \mid V = reach_\mathcal{A}(U, a) \wedge V \neq \varnothing \}$.*

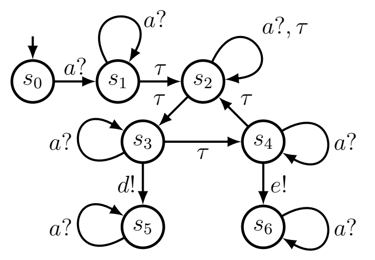
Example 2.1. The (not yet well-formed) QIOTS $\mathcal{A}$ in Figure 2(a) is nondeterministic; its determinization $det(\mathcal{A})$ is shown in Figure 2(b). □

2.2 Quiescence, fairness and divergence

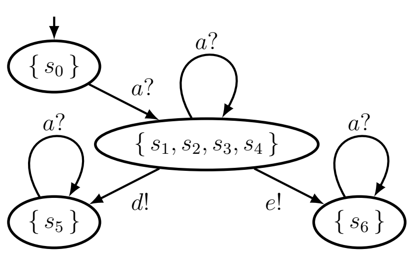
Definition 2.6 (Quiescent state). *Let $\mathcal{A}$ be a QIOTS. A state $s \in S$ is* quiescent*, denoted $q(s)$, if it has no locally-controlled actions enabled, i.e. $q(s)$ if $s \not\xrightarrow{a}$ for all $a \in L^\mathrm{O} \cup \{ \tau \}$. The set of all quiescent states of $\mathcal{A}$ is denoted $q(\mathcal{A})$.*

Example 2.2. States s_0, s_5 and s_6 of the QIOTS in Figure 2(a) are quiescent. □

As we have already discussed in the introduction, the notion of fairness plays a crucial role in the treatment of divergences in QIOTSs. As announced, we take the solution proposed in [2] for DQTSs —which in turn uses a notion of fairness that stems from [10, 3, 11]— and simplify it for our purposes. Restricted to QIOTSs, fairness states that every locally controlled action enabled from a state that is visited infinitely often, must also be executed infinitely often. Note that finite paths are fair by default.



(a) $\mathcal{A}$



(b) $det(\mathcal{A})$

Fig. 2. Visual representations of the (not yet well-formed) QIOTSs $\mathcal{A}$ and $det(\mathcal{A})$.

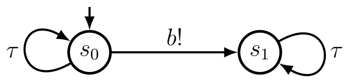


Fig. 3. A simple transition system with two types of divergence.

Definition 2.7 (Fair path). *Let $\mathcal{A}$ be a QIOTS and $\pi = s_0\, a_1\, s_1\, a_2\, s_2\, \ldots$ a path of $\mathcal{A}$. Then, π is* fair *if for all $s \in \omega\text{-}states(\pi)$ and $s \xrightarrow{a} s'$ with $a \in (L^{\mathrm{O}} \cup \{\,\tau\,\})$ the transition $s\,a\,s'$ occurs infinitely often in π.*

The set of all fair paths of a QIOTS $\mathcal{A}$ is denoted $fpaths(\mathcal{A})$, and the set of corresponding traces is denoted $ftraces(\mathcal{A})$.

Unfair paths are considered not to occur, so from now on we only consider $fpaths(\mathcal{A})$ and $ftraces(\mathcal{A})$ for the behaviour of $\mathcal{A}$.

Example 2.3. Consider the QIOTS in Figure 3. The infinite path given by $\pi = s_0\, \tau\, s_0\, \tau\, s_0\, \ldots$ is not fair as the b-output is ignored forever. □

We can now formally introduce divergence as fair infinite internal behaviour.

Definition 2.8 (Divergent path). *Let $\mathcal{A}$ be a QIOTS, then a path π is* divergent *if $\pi \in fpaths(\mathcal{A})$ and it contains only transitions labelled with the internal action τ. The set of all divergent paths of $\mathcal{A}$ is denoted $dpaths(\mathcal{A})$.*

Example 2.4. Consider the QIOTS in Figure 3 again. The infinite path given by $\pi = s_1\, \tau\, s_1\, \tau\, s_1\, \ldots$ is divergent. Note that divergent traces are not preserved by determinization. □

We are now all set to allow divergent paths to occur in QIOTSs. For computability reasons, however, we assume that each divergent path in a QIOTS contains only a finite number of states.

Definition 2.9 (State-finite path). *Let $\mathcal{A}$ be a QIOTS and let $\pi \in fpaths(\mathcal{A})$ be an infinite path. If $|states(\pi)| < \infty$, then π is* state-finite.

When the system is on a state-finite divergent path, it continuously loops through a finite number of states on this path. We call these states divergent.

Definition 2.10 (Divergent state). *Let $\mathcal{A}$ be a QIOTS. A state $s \in S$ is* divergent*, denoted $d(s)$, if there is a (state-finite and fair) divergent path on which s occurs infinitely often, i.e., if there is a path $\pi \in dpaths(\mathcal{A})$ such that $s \in \omega\text{-}states(\pi)$. The set of all divergent states of $\mathcal{A}$ is denoted $d(\mathcal{A})$.*

Divergent paths in QIOTSs may cause the observation of quiescence in states that are not necessarily quiescent themselves. As already illustrated in Figure 3, state s_1 is not quiescent, since it enables the internal action τ. Still, output is never observed on the divergent path $\pi = s_1\, \tau\, s_1\, \tau\, \ldots$, so that quiescence is observed from a non-quiescent state. Note that the assumption of strong convergence of [1] does not allow such behaviour.

2.3 Well-formedness

In Definition 2.1 we have already stipulated that, for QIOTS to be meaningful, they have to adhere to some well-formedness conditions ensuring the consistency of the representation of quiescence and divergence. Technically speaking, these conditions ensure that our QIOTSs are particular instances of the DQTS model of [2], so that we may profit from the proven properties of the more elaborate model from [2]. We first introduce the DQTS model.

Definition 2.11 (Divergent Quiescent Transition System). *A* Divergent Quiescent Transition System *(DQTS) is a tuple $\mathcal{A} = \langle\, S, S^0, L^{\mathrm{I}}, L^{\mathrm{O}}, L^{\mathrm{H}}, P, \rightarrow \rangle$, where S is a set of states; $S^0 \subseteq S$ is a non-empty set of initial states; L^{I}, L^{O} and L^{H} are disjoint sets of input, output and internal labels, respectively; P is a partition of $L^{\mathrm{O}} \cup L^{\mathrm{H}}$; and $\rightarrow\ \subseteq S \times L \times S$ is the transition relation, where $L = L^{\mathrm{I}} \cup L^{\mathrm{O}} \cup L^{\mathrm{H}} \cup \{\,\delta\,\}$. We assume $\delta \notin (L^{\mathrm{I}} \cup L^{\mathrm{O}} \cup L^{\mathrm{H}})$.*

1. *A DQTS $\mathcal{A}$ must be* input-enabled*, i.e. for each $s \in S$ and $a \in L^{\mathrm{I}}$, there exists an $s' \in S$ such that $(s, a, s') \in \rightarrow$.*
2. *A DQTS must be well-formed, i.e. it fulfils Rules R1, R2, R3, and R4 stipulated below.*

We use the notions $q(s)$ and $d(s)$, defined earlier for QIOTSs, for DQTSs as well in the obvious way.

Definition 2.12 (Well-formedness). *A DQTS (or a QIOTS) $\mathcal{A}$ is* well-formed *if it satisfies the following rules for all $s, s', s'' \in S$ and $a \in L^{\mathrm{I}}$:*

Rule R1 (Quiescence is observable)**:** if $q(s)$ or $d(s)$, then $s \xrightarrow{\delta}$.

Rule R2 (Quiescence follows quiescence)**:** if $s \xrightarrow{\delta} s'$, then $q(s')$.
We require the observation of quiescence to result in a quiescent state. This makes sure that no outputs are observed following a quiescence observation. We could have been a bit more liberal for QIOTSs, replacing this rule by "if $s \xrightarrow{\delta} s'$, then $q(s') \vee d(s')$". After all, divergent states can never invisibly reach any output transitions under our current fairness assumption. However, we chose to be more strict here, requiring traditional quiescence (the absence of any locally controlled actions) following a δ-transition. That way, these rules still work in the presence of a more liberal fairness assumption that also allows output actions to be enabled from divergent states (as in [2]).

Rule R3 (Quiescence enables no new behaviour)**:** if $s \xrightarrow{\delta} s'$, then $\mathit{traces}(s') \subseteq \mathit{traces}(s)$.
The observation of quiescence must not lead to actions that were not enabled before its observation. As the observation of quiescence may be the result of an earlier nondeterministic choice, its observation may lead to a state with fewer enabled actions.

Rule R4 (Repeated quiescence preserves behaviour)**:** if $s \xrightarrow{\delta} s'$ and $s' \xrightarrow{\delta} s''$, then $\mathit{traces}(s'') = \mathit{traces}(s')$.

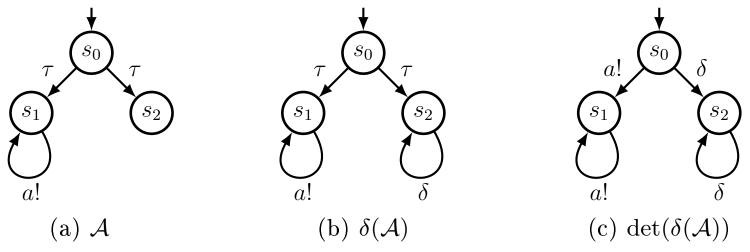


Fig. 4. Illustration of Rule R3.

The potential reduction of enabled actions of the previous clause only manifests itself in the first observation of quiescence; its continued observation adds no new information.

For finite DQTSs/QIOTSs the validity of all rules is computable. Further below we will discuss the computation of $d(s)$.

Example 2.5. To illustrate the necessity of using $\subseteq$ instead of $=$ in Rule R3, observe the (not yet well-formed) nondeterministic system in Figure 4(a). To make it follow Rule R1, we added a δ-transition to state s_2, as shown in Figure 4(b). Then, we determinized the system, obtaining the QIOTS shown in Figure 4(c). All rules are satisfied. Indeed, as allowed by Rule R3, *traces*(s_2) is a proper subset of *traces*(s_0). □

2.4 QIOTSs versus DQTSs

Looking at the definition of DQTSs we observe a few differences with QIOTSs. As the definition of DQTSs is based on the Input-Output Automata model of Lynch and Tuttle [3], the three differences are that there is a set of initial states, rather than a single state; that there is a set of named internal actions, rather than just τ; and that a partition P of $L^{\mathrm{O}} \cup L^{\mathrm{H}}$ is part of the model. Concerning the first two differences, the QIOTS model is just a straightforward special case of the DQTS model. The partition P represents the fairness conditions that apply to the model, which can be tuned as part of the model. The definition of a fair path for a DQTS is as follows: a path $\pi = s_0\, a_1\, s_1\, a_2\, s_2 \ldots$ is fair if for every $A \in P$ such that $\exists^\infty j\ .\ L(s_j) \cap A \neq \varnothing$, we have $\exists^\infty j\ .\ a_j \in A$. As for QIOTSs, state-finite divergence is assumed. The definition below shows which DQTSs correspond to QIOTSs.

Definition 2.13 (Associated DQTS). *Let $\mathcal{A} = \langle\, S, s^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$ be a QIOTS, then its* associated Divergent Quiescent Transition System *$DQTS(\mathcal{A})$ is defined by the tuple $\langle\, S, \{\, s^0 \,\}, L^{\mathrm{I}}, L^{\mathrm{O}}, L^{\mathrm{H}}, P, \rightarrow' \rangle$ with*

1. $L^{\mathrm{H}} = \{\, \tau_{(s,s')} \mid s \xrightarrow{\tau} s' \in \rightarrow \};$

2. $P = \{\, \{a\} \mid a \in L^{\mathrm{O}} \cup L^{\mathrm{H}} \,\}$
3. $\rightarrow' = \rightarrow \setminus \{\, (s,\tau,s') \mid (s,\tau,s') \in \rightarrow \,\} \cup \{\, (s,\tau_{(s,s')},s') \mid (s,\tau,s') \in \rightarrow \,\}$

It is straightforward to check that this association preserves the intended fairness condition for QIOTSs. In [2] it is proven that well-formed DQTSs and SAs are equivalent in terms of expressible observable behaviour: it is shown that for every DQTS there exists a trace equivalent SA, and vice versa. Hence, except for divergences, their expressivity coincides. This result carries over to QIOTSs, as their restriction with respect to DQTSs does not affect the observable traces.

3 Operations and properties

3.1 Deltafication: from IOTS to QIOTS

Usually, specifications are not modelled as QIOTSs directly, but rather in a formalism whose operational semantics can be expressed in terms of IOTSs. Hence, we need a way to convert an IOTS to a (well-formed) QIOTS that captures all possible observations of it, including quiescence. This conversion is called *deltafication*. As for QIOTSs, we require all IOTSs to be input-enabled for deltafication.

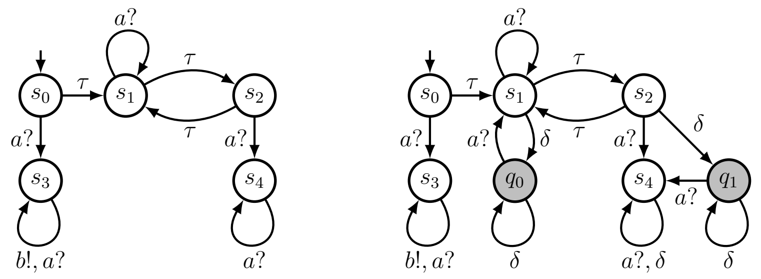
To satisfy rule R1, every state in which quiescence may be observed (i.e., all quiescent and divergent states) must have an outgoing δ-transition. Hence, to go from an IOTS to a QIOTS, the deltafication operator adds a δ-labelled self-loop to each quiescent state. Also, a new *quiescence observation state* qos_s is introduced for each divergent state $s \in S$: When quiescence is observed in s, a δ-transition will lead to qos_s. To preserve the original behaviour, inputs from qos_s must lead to the same states that the corresponding input transitions from s led to. All these considerations together lead to the following definition for the deltafication procedure for IOTSs.

As mentioned before, the SA construction that adds δ-labelled self-loops to all quiescent states does not work for divergent states, since divergent states must have at least one outgoing internal transition (and possibly even output transitions when taking a more lenient fairness assumption, as in [2]). So, a δ-labelled self-loop added to a divergent state would contradict rule R2.

Definition 3.1 (Deltafication). *Let $\mathcal{A} = \langle S_{\mathcal{A}}, s^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow_{\mathcal{A}} \rangle$ be an IOTS with $\delta \notin L$. The* deltafication *of $\mathcal{A}$ is $\delta(\mathcal{A}) = \langle S_\delta, s^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow_\delta \rangle$. We define $S_\delta = S_{\mathcal{A}} \cup \{\, qos_s \mid s \in d(\mathcal{A}) \,\}$, i.e., S_δ contains a new state $qos_s \notin S_{\mathcal{A}}$ for every divergent state $s \in S_{\mathcal{A}}$ of $\mathcal{A}$. The transition relation $\rightarrow_\delta$ is as follows:*

$$\begin{aligned}
\rightarrow_\delta = \rightarrow_{\mathcal{A}} \;&\cup\; \{\, (s,\delta,s) \quad\quad \mid s \in q(\mathcal{A}) \,\} \\
&\cup\; \{\, (s,\delta,qos_s) \;\mid s \in d(\mathcal{A}) \,\} \cup \{\, (qos_s,\delta,qos_s) \mid s \in d(\mathcal{A}) \,\} \\
&\cup\; \{\, (qos_s,a?,s') \mid s \in d(\mathcal{A}) \;\wedge\; a? \in L^{\mathrm{I}} \;\wedge\; s \xrightarrow{a?}_{\mathcal{A}} s' \,\}
\end{aligned}$$

Note that computing $q(\mathcal{A})$ is trivial: simply identify all states without outgoing output or internal transitions. Determining $d(\mathcal{A})$ is discussed further below.



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

Fig. 5. An IOTS $\mathcal{A}$ and its deltafication $\delta(\mathcal{A})$. Newly introduced states are grey.

Example 3.1. See Figure 5 for IOTS $\mathcal{A}$ and its deltafication. States s_1 and s_2 are divergent, and q_0 and q_1 quiescence observation states. Note that s_0 has an outgoing divergent path, while in accordance to rule R1 it is not given an outgoing δ-transition. The reason is that, when observing quiescence due to divergence, rule R4 prescribes that the system can only reside in s_1 or s_2. The states reachable from a given state via unobservable paths assume a similar role as the *stable* states (not having outgoing τ-transitions) in [1], which does not deal with divergence. So, quiescence cannot be observed from s_0, and therefore also the a-transition to s_3 should not be enabled anymore after observation of quiescence. This is now taken care of by not having a direct δ-transition from s_0. Because of this, no trace first having δ and then the $b!$ output is present. $\square$

The results from [2] imply directly that the deltafication $\delta(\mathcal{A})$ indeed yields a well-formed QIOTS for every IOTS $\mathcal{A}$.

In order to compute the set of divergent states $d(\mathcal{A})$ in a QIOTS $\mathcal{A}$, we proceed as follows. First, we mark all states that enable an output action; say we colour those states red. Then, we consider the directed graph G that we obtain from $\mathcal{A}$ by keeping only the τ transitions, and removing all transitions labelled by an input or output action. Thus, $G = (S, E)$ with $E = \{(s, s') \mid s \xrightarrow{\tau} s'\}$. In G, we compute, using Tarjan's algorithm [12], the set of all bottom strongly connected components. Now, a state is divergent if and only if it is contained in a bottom strongly connected component that contains no red state and has at least one τ-transition.

3.2 Composition of QIOTSs

Parallel composition is an important standard operation on well-formed QIOTSs, and again is a straightforward specialization of the corresponding definition for DQTSs in [2]. To apply the parallel composition operator we require every output action to be under the control of at most one component [3].

Definition 3.2 (Compatibility). *Two QIOTSs $\mathcal{A}$ and $\mathcal{B}$ are* compatible *if $L^{\mathrm{O}}_{\mathcal{A}} \cap L^{\mathrm{O}}_{\mathcal{B}} = \varnothing$.*

Definition 3.3 (Parallel composition). *Given two well-formed compatible QIOTSs $\mathcal{A}$ and $\mathcal{B}$, the* parallel composition *of $\mathcal{A}$ and $\mathcal{B}$ is the QIOTS $\mathcal{A} \parallel \mathcal{B}$, with $S_{\mathcal{A}\|\mathcal{B}} = S_{\mathcal{A}} \times S_{\mathcal{B}}$, $s^0_{\mathcal{A}\|\mathcal{B}} = (s^0_{\mathcal{A}}, s^0_{\mathcal{B}})$, $L^{\mathrm{I}}_{\mathcal{A}\|\mathcal{B}} = (L^{\mathrm{I}}_{\mathcal{A}} \cup L^{\mathrm{I}}_{\mathcal{B}}) \setminus (L^{\mathrm{O}}_{\mathcal{A}} \cup L^{\mathrm{O}}_{\mathcal{B}})$, $L^{\mathrm{O}}_{\mathcal{A}\|\mathcal{B}} = L^{\mathrm{O}}_{\mathcal{A}} \cup L^{\mathrm{O}}_{\mathcal{B}}$, and*

$$\begin{aligned}
\rightarrow_{\mathcal{A}\|\mathcal{B}} = \; & \{\, ((s,t),a,(s',t')) \in S_{\mathcal{A}\|\mathcal{B}} \times ((L_{\mathcal{A}} \cap L_{\mathcal{B}}) \setminus \{\,\tau\,\}) \times S_{\mathcal{A}\|\mathcal{B}} \mid \\
& \quad s \xrightarrow{a}_{\mathcal{A}} s' \;\wedge\; t \xrightarrow{a}_{\mathcal{B}} t' \,\} \\
\cup \; & \{\, ((s,t),a,(s',t)) \in S_{\mathcal{A}\|\mathcal{B}} \times (L_{\mathcal{A}} \setminus (L_{\mathcal{B}} \setminus \{\,\tau\,\})) \times S_{\mathcal{A}\|\mathcal{B}} \mid s \xrightarrow{a}_{\mathcal{A}} s' \,\} \\
\cup \; & \{\, ((s,t),a,(s,t')) \in S_{\mathcal{A}\|\mathcal{B}} \times (L_{\mathcal{B}} \setminus (L_{\mathcal{A}} \setminus \{\,\tau\,\})) \times S_{\mathcal{A}\|\mathcal{B}} \mid t \xrightarrow{a}_{\mathcal{B}} t' \,\}
\end{aligned}$$

We have $L_{\mathcal{A}\|\mathcal{B}} = L^{\mathrm{I}}_{\mathcal{A}\|\mathcal{B}} \cup L^{\mathrm{O}}_{\mathcal{A}\|\mathcal{B}} \cup \{\,\tau\,\} = L_{\mathcal{A}} \cup L_{\mathcal{B}}$.

In essence this is the usual process algebraic definition of parallel composition with synchronization on shared labels (first set of transitions in the definition of $\rightarrow_{\mathcal{A}\|\mathcal{B}}$) and interleaving on independent labels (second and third sets of transitions in the definition of $\rightarrow_{\mathcal{A}\|\mathcal{B}}$). Note that δ is a shared label that must synchronize, and that τ never synchronizes.

3.3 Preservation properties

In [2] it is shown that well-formed DQTSs are preserved under parallel composition and determinization. These results carry over directly to QIOTSs. This is only possible because of the refined treatment of the definition of quiescence.

With the representation of quiescence by simple δ-loops, as it is done in the SA model, preservation under determinization fails. It requires that δ-loops can be unwound, because of the need to preserve Rule R2 under determinization.

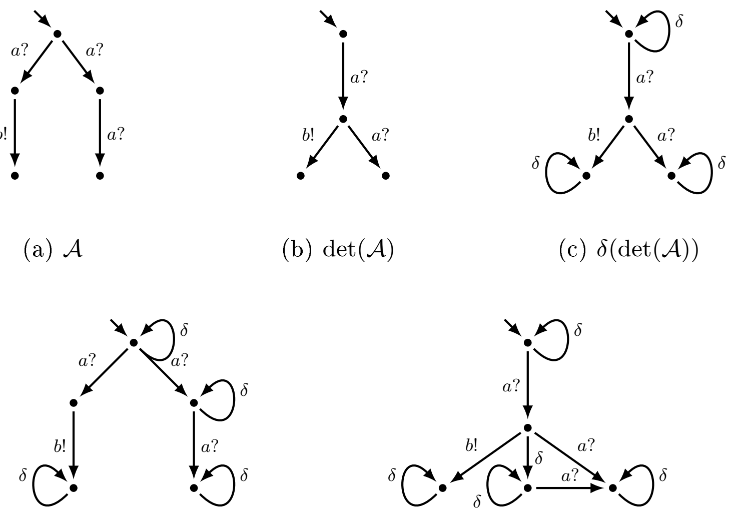
Another crucial operation (on IOTSs) is deltafication. A pleasing result is that deltafication and parallel composition commute [2]. That is, given two compatible IOTSs $\mathcal{A}$, $\mathcal{B}$, such that $\delta \notin L_{\mathcal{A}} \cup L_{\mathcal{B}}$, we have $\delta(\mathcal{A} \parallel \mathcal{B}) \;\approx_{\mathrm{tr}}\; \delta(\mathcal{A}) \parallel \delta(\mathcal{B})$.

With deltafication and determinization the situation is more involved. This is a direct consequence of the fact that determinization does not preserve quiescence. Of course, determinization removes divergences by construction, but this is not the only source of problems, as the example in Figure 6 shows (omitting some self-loops needed for input-enabledness, for presentation purposes).

It follows that when transforming a nondeterministic IOTS $\mathcal{A}$ to a deterministic, well-formed QIOTS, one should always first derive $\delta(\mathcal{A})$ and only then obtain a determinized version. As demonstrated in Figure 6(e), self-loops labelled δ may turn into regular transitions, motivating once more our choice of moving away from Suspension Automata (that were not closed under determinization) to a more general framework in which δ-transitions are treated as first-class citizens.

3.4 Conformance for QIOTSs

The core of the `ioco`-framework is its *conformance relation*, relating specifications to implementations if and only if the latter is 'correct' with respect to the former. For `ioco`, this means that the implementation never provides an



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

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

Fig. 6. Determinization and deltafication.

unexpected output (including quiescence) when it is only fed inputs that are allowed by the specification. Traditionally, this was formalized based on the SAs corresponding to the implementation and the specification. Now, we can apply well-formed QIOTSs, as they already model the expected absence of outputs by explicit δ-transitions. As QIOTSs support divergence, using them also allows `ioco` to be applied in the presence of (finite state) divergence.

Definition 3.4 (`ioco`)**.** *Let* $\mathsf{Impl}, \mathsf{Spec}$ *be well-formed QIOTSs over the same alphabet. Then,* $\mathsf{Impl} \sqsubseteq_{\mathtt{ioco}} \mathsf{Spec}$ *if and only if*

$$\forall \sigma \in \mathit{traces}(\mathsf{Spec}) \ . \ \mathit{out}_{\mathsf{Impl}}(\sigma) \subseteq \mathit{out}_{\mathsf{Spec}}(\sigma),$$

where $\mathit{out}_{\mathcal{A}}(\sigma) = \{a \in L^{\mathrm{O}}_{\delta} \mid \sigma a \in \mathit{traces}(\mathcal{A})\}$.

This new notion of `ioco`-conformance can be applied to extend the testing frameworks in [8, 5], using the same basic schema: during testing, continuously choose to either try and provide an input, observe the behaviour of the system or stop testing. As long as the trace obtained this way, including the δ actions as the result of either quiescence or divergence, is also a trace of the specification, the implementation is correct.

Since all QIOTSs are required to be input-enabled, it is easy to see that `ioco`-conformance precisely corresponds to traditional trace inclusion over well-formed QIOTSs (and hence, $\sqsubseteq_{\mathtt{ioco}}$ is transitive). Note that this only holds because the specification Spec and implementation Impl are already represented as QIOTSs;

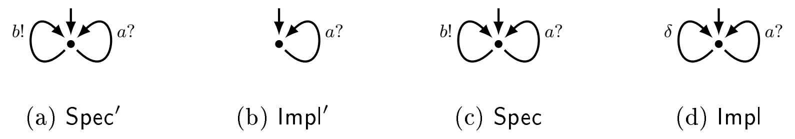


Fig. 7. Illustration of Example 3.2.

trace inclusion of the IOTSs Impl' and Spec' from which these QIOTSs may have been generated does not necessarily imply that $\mathsf{Impl} \sqsubseteq_{\texttt{ioco}} \mathsf{Spec}$. The following example illustrates this.

Example 3.2. Consider the systems shown in Figure 7, all over the action signature $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta}) = (\{a?\}, \{b!, \delta\})$. Clearly, both the IOTSs Spec' and Impl' are input-enabled, and also $\mathit{traces}(\mathsf{Impl}') \subseteq \mathit{traces}(\mathsf{Spec}')$. However, when looking at the corresponding QIOTSs Spec and Impl, we see that $\delta \in \mathit{out}_{\mathsf{Impl}}(\epsilon)$, but $\delta \notin \mathit{out}_{\mathsf{Spec}}(\epsilon)$. Therefore, $\mathit{out}_{\mathsf{Impl}}(\epsilon) \not\subseteq \mathit{out}_{\mathsf{Spec}}(\epsilon)$, and as $\epsilon \in \mathit{traces}(\mathsf{Spec})$ by definition $\mathsf{Impl} \not\sqsubseteq_{\texttt{ioco}} \mathsf{Spec}$. □

Clearly, action hiding—renaming output actions to τ—does not necessarily preserve $\sqsubseteq_{\texttt{ioco}}$. After all, it may introduce quiescence where that is not allowed by the specification. (Note also that δ-transitions may need to be added after hiding. We refer to [2] for a detailed exposition of the hiding operator on DQTSs.)

4 Test cases and test suites

We present part of the testing framework introduced in [5], applying it to QIOTSs instead of the more basic QTSs used there. Whether the δ-transitions in QIOTSs were introduced because of traditional quiescence or divergence does not influence their behaviour, so all results from [5] still hold and the proofs are not all repeated here.

4.1 Tests over an action signature

We apply model-based testing in a *black-box* manner: to execute a test case on a system, one only needs an executable of the implementation. Hence, test cases and test suites can be defined solely based on the so-called *action signature* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$, also called *interface*, of the implementation.

A test case describes the behaviour of a tester. At each moment in time the tester either stops, or waits for the system to do something, or tries to provide an input. This is represented for each trace (a *history*) σ in the test case having either (1) no other traces of which σ is a prefix, (2) several traces $\sigma b!$ that extend σ with all output actions from L^{O}_{δ}, or (3) a single trace $\sigma a?$ extending σ with an input action $a? \in L^{\mathrm{I}}$. In the third case, there should also be traces $\sigma b!$ for all actions $b! \in L^{\mathrm{O}}$ (excluding δ), as the implementation may be faster than the tester. A test case contains all behaviour that *may* occur during testing — during an actual test, however, only one complete trace of the test will be observed.

Definition 4.1 (Test case). *A* test case *(or shortly a* test*) over an action signature* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ *is a set of traces* $t \subseteq (L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta})^*$ *such that*

- *t is prefix-closed;*
- *t does not contain an infinite increasing sequence $\sigma_0 \sqsubset \sigma_1 \sqsubset \sigma_2 \sqsubset \dots$*[1]*;*
- *For every trace $\sigma \in t$, we have either*
  1. *$\{a \in L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta} \mid \sigma a \in t\} = \varnothing$, or*
  2. *$\{a \in L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta} \mid \sigma a \in t\} = L^{\mathrm{O}}_{\delta}$, or*
  3. *$\{a \in L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta} \mid \sigma a \in t\} = \{a?\} \cup L^{\mathrm{O}}$ for some $a? \in L^{\mathrm{I}}$.*

Test cases should not contain an infinite trace $aaa\dots$ (taken care of by using $(L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta})^*$ instead of $(L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta})^{\infty}$) or an infinite increasing sequence $a, aa, aaa, \dots$ (taken care of by the second condition), since we want the test process to end at some point. By requiring them to adhere to the observations made above on the type of traces that they contain, they necessarily represent a deterministic tester (i.e., they never nondeterministically choose between different input actions).

We note that test cases can be represented as directed acyclic graphs (DAGs) as well.

Definition 4.2 (Test case notations). *Given an action signature* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*,*

- *we use $\mathcal{T}(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ to denote the set of all tests over $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$.*
- *we define a* test suite *over $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ to be a* set *of tests over $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$. We denote the set of all test suites over $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ by $\mathcal{TS}(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$.*

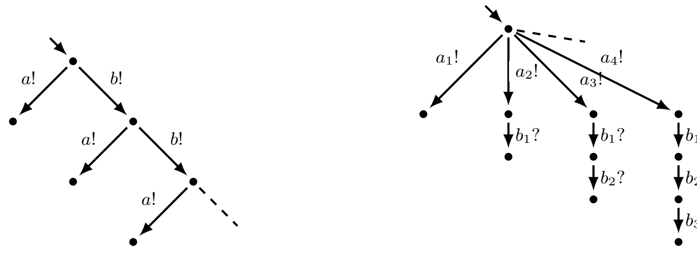
Given a test case t over $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$,

- *we say that the* length *of t is the supremum of the lengths of the traces in t, i.e., $\sup\{|\sigma| \mid \sigma \in t\}$. Note that this length is an element of $\mathbb{N} \cup \{\infty\}$.*
- *we say that t is* linear *if there exists a trace $\sigma \in t$ such that every non-empty trace $\rho \in t$ can be written as $\sigma' a$, where $\sigma' \sqsubseteq \sigma$ and $a \in L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta}$. The trace σ is called the* main trace *of t.*
- *we use $ctraces(t)$ to denote the* complete traces *of t, i.e., all traces $\sigma \in t$ for which there is no $\rho \in t$ such that $\sigma \sqsubset \rho$.*

Example 4.1. The restriction that a test case cannot contain an infinite increasing sequence makes sure that every test process will eventually terminate. However, it does not mean that the length of a test case is necessarily finite.

To see this, observe the two tests shown in Figure 8 (represented as DAGs, and for presentation purposes not showing all transitions). The DAG shown in Figure 8(a) is not allowed, as it contains the infinite path $b!\, b!\, b!\, b! \dots$. Therefore, a test process based on this DAG may never end. The DAG shown in Figure 8(b), however, is a valid test. Although it has infinite length (after all, there

[1] If $L^{\mathrm{I}} \cup L^{\mathrm{O}}_{\delta}$ is finite, we can replace this requirement by asking that t is finite.



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

Fig. 8. Infinite tests.

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 8(a) at a certain depth is linear, whereas the test in Figure 8(b) is not. $\Box$

Definition 4.3 (Tests for a specification). *Let* $\mathsf{Spec} = \langle S, s^0, L^{\mathrm{I}}, L^{\mathrm{O}}, \rightarrow \rangle$ *be a specification (i.e., a QIOTS), then a test for* Spec *is a test over* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*. We denote the universe of tests and test suites for* Spec *by* $\mathcal{T}(\mathsf{Spec})$ *and* $\mathcal{TS}(\mathsf{Spec})$*, respectively.*

4.2 Test annotations, executions and verdicts

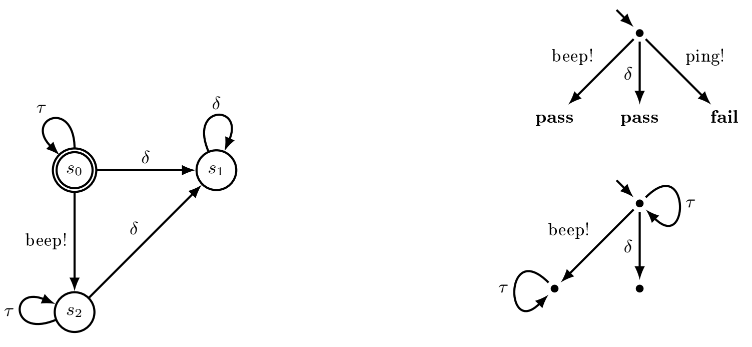
Before testing a system, we obviously need to define which outcomes of a test case are considered correct (the system *passes*), and which are considered incorrect (the system *fails*). For this purpose we introduce *annotations*.

Definition 4.4 (Annotations). *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* $\widehat{T} = \{(t_i, a_i)\}$ *is called an* annotated test suite.

When representing a test case as DAG, we depict the annotation function by means of labels on its leaves (see Figure 9(b)).

Running a test case can basically be considered as the parallel composition of the test and the implementation, after first mirroring the action labels of the test for synchronisation to take place (changing inputs into outputs and the other way around)[2]. Note that the test and the implementation synchronise on all visible actions, that the implementation cannot block any inputs and that the

[2] Technically, parallel composition was only defined for QIOTSs, and test cases are no QIOTSs. However, the idea can easily be lifted. Moreover, the actual formal definition of the execution of a test case below circumvents this issue by directly defining the results of the parallel composition.



(a) Implementation Impl (b) Test case t and parallel composition.

Fig. 9. An implementation, test case and their parallel composition.

test cannot block any outputs (except at the end). Therefore, it can easily be seen that the set of possible traces arising from this parallel composition is just the intersection of the trace sets of the test and the implementation. We are mainly interested in the complete traces of this intersection, as they contain the most information. Also, we prefer to exclude the empty trace, as it cannot be observed during testing anyway (rather, it could be observed by means of a δ-transition). To accommodate this, we directly define the set of possible *executions* of a test case t given an implementation Impl as follows.

Definition 4.5 (Executions). *Let* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ *be an action signature,* t *a test case over* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*, and* Impl *an QIOTS over* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*. Then,*

$$exec_t(\mathsf{Impl}) = traces(\mathsf{Impl}) \cap ctraces(t)$$

are the executions *of* t *given* Impl.

Example 4.2. Consider the implementation in Figure 9(a) and the corresponding test case in Figure 9(b). Figure 9(b) additionally shows their parallel composition (after first mirroring the test case). Note that it is immediately clear from this parallel composition that the erroneous output *ping*! is not present in the implementation. By definition, the executions of this test case t given the implemention Impl are $exec_t(\mathsf{Impl}) = \{\text{beep!}, \delta\}$. □

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 $ctraces(t)$ and was annotated by *fail*), and the verdict *fail* is given otherwise.

Definition 4.6 (Verdict functions). *Let* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ *be an action signature and* $\hat{t} = (t, a)$ *an annotated test case over* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*. The* verdict function *for* $\hat{t}$ *is the function*

$$v_{\hat{t}} \colon \mathcal{QIOTS}(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta}) \to \{pass, fail\},$$

given for any QIOTS Impl *by*

$$v_{\hat{t}}(\mathsf{Impl}) = \begin{cases} \textit{pass} & \textit{if } \forall\, \sigma \in \textit{exec}_t(\mathsf{Impl}) \;.\; a(\sigma) = \textit{pass}; \\ \textit{fail} & \textit{otherwise.} \end{cases}$$

We extend $v_{\hat{t}}$ *to a function* $v_{\widehat{T}} \colon \mathcal{QIOTS}(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta}) \to \{\textit{pass}, \textit{fail}\}$ *assigning a verdict to implementations based on a test suite, by putting* $v_{\widehat{T}}(\mathsf{Impl}) = \textit{pass}$ *if* $v_{\hat{t}}(\mathsf{Impl}) = \textit{pass}$ *for all* $\hat{t} \in \widehat{T}$*, and* $v_{\widehat{T}}(\mathsf{Impl}) = \textit{fail}$ *otherwise.*

Remark 4.1. Note that during (and after) testing we only have a partial view of the set $\textit{exec}_t(\mathsf{Impl})$, and hence of Impl: each time we run a test case on an implementation, we see only one single trace in the test case. Additionally, we are not in control of which traces we see. It is the implementation that decides which branch is selected. Hence, each test case should be executed a number of times to cover all behaviour. This implies that testing is inherently incomplete; even though no failure has been observed, there still may be faults left in the system. □

5 Testing with respect to $\sqsubseteq_{\mathtt{ioco}}$

The conformance relation $\sqsubseteq_{\mathtt{ioco}}$ is of vital importance in the test process, as it captures precisely which behaviour is considered valid and which is considered invalid. Based on Definition 3.4, it induces the following annotation function:

Definition 5.1 (ioco-annotation). *Let t be an (unannotated) test case for a specification* Spec*. The annotation function* $a^{\mathtt{ioco}}_{\mathsf{Spec},t} \colon \textit{ctraces}(t) \to \{\textit{pass}, \textit{fail}\}$ *for t is given by*

$$a^{\mathtt{ioco}}_{\mathsf{Spec},t}(\sigma) = \begin{cases} \textit{fail} \;\; \textit{if } \exists\, \sigma_1 \in \textit{traces}(\mathsf{Spec}), a! \in L^{\mathrm{O}}_{\delta} \;. \\ \qquad\quad \sigma \sqsupseteq \sigma_1 a! \wedge \sigma_1 a! \notin \textit{traces}(\mathsf{Spec}); \\ \textit{pass otherwise.} \end{cases}$$

The basic idea is that we generally assign a *fail* verdict only to sequences σ that can be written as $\sigma = \sigma_1 a! \sigma_2$ such that $\sigma_1 \in \textit{traces}(\mathsf{Spec})$ and $\sigma_1 a! \notin \textit{traces}(\mathsf{Spec})$; that is, when there is an output action that leads us out of the traces of Spec. Note that if we can write $\sigma = \sigma_1 b? \sigma_2$ such that $\sigma_1 \in \textit{traces}(\mathsf{Spec})$ and $\sigma_1 b? \notin \textit{traces}(\mathsf{Spec})$, then we assign a *pass*, because in this case an unexpected input $b? \in L^{\mathrm{I}}$ was provided by the test case. Hence, any behaviour that comes after this input is `ioco`-conforming.

Remark 5.1. In our setting of input-enabled specifications, the scenario in which $\sigma = \sigma_1 b? \sigma_2$ such that $\sigma_1 \in \textit{traces}(\mathsf{Spec})$ and $\sigma_1 b? \notin \textit{traces}(\mathsf{Spec})$, cannot occur. The definition reduces to $a^{\mathtt{ioco}}_{\mathsf{Spec},t}(\sigma) = \textit{pass}$ if and only if $\sigma \in \textit{traces}(\mathsf{Spec})$. □

Example 5.1. In Figure 10, test case t_3 is annotated according to $a^{\mathtt{ioco}}_{\mathsf{Spec},t_3}$. Test case t_1 is not, though, since it should allow the trace $a!$ and not the trace $b!$ (since $\epsilon \in \textit{traces}(\mathsf{Spec})$, $b! \in L^{\mathrm{O}}_{\delta}$ and $b! \notin \textit{traces}(\mathsf{Spec})$). Test case t_2 is also not annotated according to $a^{\mathtt{ioco}}_{\mathsf{Spec},t_3}$, since it erroneously allows δ. □

Given a specification Spec, any test case t annotated according to $a^{\mathtt{ioco}}_{\mathsf{Spec},t}$ is *sound* for Spec with respect to $\sqsubseteq_{\mathtt{ioco}}$. Intuitively, a sound test case never rejects a correct implementation. That is, for all implementations $\mathsf{Impl} \in \mathcal{QIOTS}(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ it holds that $v_{\hat{t}}(\mathsf{Impl}) = \mathit{fail}$ implies $\mathsf{Impl} \not\sqsubseteq_{\mathtt{ioco}} \mathsf{Spec}$. It is easy to see that, as $\sqsubseteq_{\mathtt{ioco}}$ coincides with trace inclusion due to input-enabledness, a test case is sound if $\forall\, \sigma \in \mathit{ctraces}(t) \,.\, \sigma \in \mathit{traces}(\mathsf{Spec}) \implies a(\sigma) = \mathit{pass}$.

The fact that $a^{\mathtt{ioco}}_{\mathsf{Spec},t}$ yields sound test cases follows directly from the above observation and Remark 5.1.

Proposition 5.1. *Let* Spec *be a specification, then the annotated test suite* $\widehat{T} = \{(t, a^{\mathtt{ioco}}_{\mathsf{Spec},t}) \mid t \in \mathcal{T}(\mathsf{Spec})\}$ *is sound for* Spec *with respect to* $\sqsubseteq_{\mathtt{ioco}}$.

To also state a completeness property we first introduce a canonical form for sequences, based on the idea that it is never needed to test for quiescence multiple times consecutively.

Definition 5.2 (Canonical traces). *Let σ be a sequence over a label set L with $\delta \in L$, then its canonical form $\mathit{canon}(\sigma)$ is the sequence obtained by replacing every occurrence 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*

$$\mathit{canon}(S) = \{\mathit{canon}(\sigma) \mid \sigma \in S\}.$$

The following proposition precisely characterises the requirement for a test suite to be complete with respect to $\sqsubseteq_{\mathtt{ioco}}$. Intuitively, a complete test suite never accepts an incorrect implementation. That is, $\widehat{T}$ is *complete* for Spec with respect to $\sqsubseteq_{\mathtt{ioco}}$ if for all implementations $\mathsf{Impl} \in \mathcal{QIOTS}(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$ it holds that $\mathsf{Impl} \not\sqsubseteq_{\mathtt{ioco}} \mathsf{Spec} \implies v_{\widehat{T}}(\mathsf{Impl}) = \mathit{fail}$.

Proposition 5.2. *Given a specification* Spec *and a test suite* $\widehat{T} \subseteq \{(t, a^{\mathtt{ioco}}_{\mathsf{Spec},t}) \mid t \in \mathcal{T}(\mathsf{Spec})\}$, $\widehat{T}$ *is complete for* Spec *with respect to* $\sqsubseteq_{\mathtt{ioco}}$ *if and only if*

$$\begin{gathered}\forall\, \sigma \in \mathit{canon}(\mathit{traces}(\mathsf{Spec})) \,. \\ \big(\mathit{out}_{\mathsf{Spec}}(\sigma) \neq L^{\mathrm{O}}_{\delta} \implies \exists\, (t, a) \in \widehat{T} \,.\, \sigma\delta \in t\big)\end{gathered}$$

Proof (sketch). This proposition states that a complete test suite should be able to observe the implementation's behaviour following every canonical trace of the specification (except when all behaviour is allowed). Hence, no possible unexpected (erroneous) outputs are impossible to detect, and indeed every incorrect implementation can be caught. The fact that we can restrict to canonical traces stems from well-formedness rules R2 and R4, which make sure that it is never necessary to directly observe again after observing quiescence. □

Example 5.2. Consider the specification, implementation and test cases shown in Figure 10, all assuming $L^{\mathrm{O}} = \{a!, b!\}$ and $L^{\mathrm{I}} = \varnothing$. Note that $\mathsf{Impl} \not\sqsubseteq_{\mathtt{ioco}} \mathsf{Spec}$ due to the unexpected $b!$ output.

Test case t_1 is not sound for Spec with respect to $\sqsubseteq_{\mathtt{ioco}}$, since it fails the correct implementation Spec. Test case t_2 is sound, though, as it only rejects

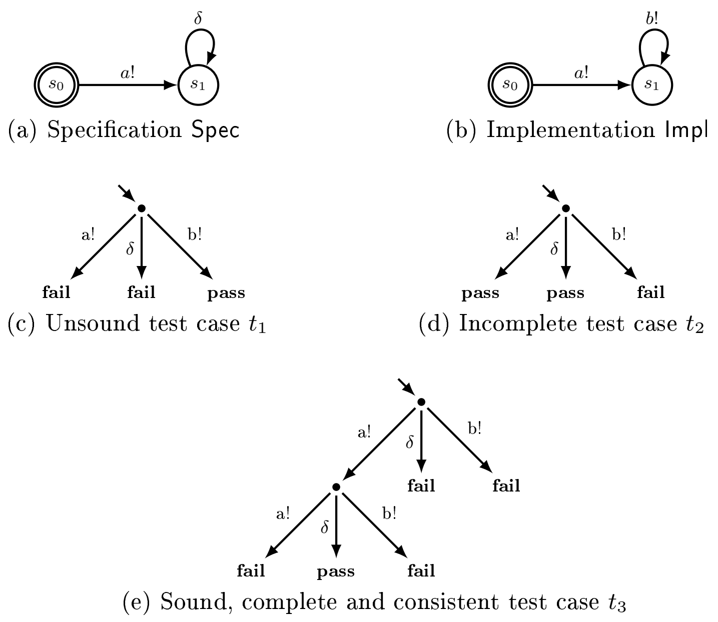


Fig. 10. A specification, implementation and test cases.

implementations that start with an unexpected $b!$ (as not allowed by $\sqsubseteq_{\texttt{ioco}}$). Although t_2 is sound, it is not complete; it does not detect Impl to be erroneous, since it stops testing after the first transition.

Using the characterisation of completeness, we can now easily show that each test suite containing the test case t_3 is complete for Spec with respect to $\sqsubseteq_{\texttt{ioco}}$. After all, $canon(traces(\mathsf{Spec})) = \{\epsilon, a!\}$, and indeed $\delta \in t_3$ and $a!\delta \in t_3$. Note that we can indeed stop testing after the δ observation, since the well-formedness rules of QIOTSs do not allow any outputs after a δ transition. □

Soundness is a necessary, but not a sufficient property for an annotated test case to be useful. Indeed, a test case annotated with only pass verdicts is always sound. Therefore, we prefer a test case to give a fail verdict whenever it should, i.e., whenever its execution with an implementation produces a trace that is not allowed by the specification. Of course, completeness of a test suite makes sure that such traces are failed by at least one test case in the suite, but that is not necessarily efficient, and moreover, full completeness is rarely achievable. In practice, there are two reasons why testing is always incomplete: first, a complete test suite typically has infinitely many test cases, whereas we can execute only finitely many of them. Second, as observed in Remark 4.1, executing one test case yields only a partial view of the implementation, as each test execution reveals a single trace from the test. Therefore, we propose the more local notion of consistency, extending soundness by requiring that implementations should not pass a test case that observes behaviour that is not allowed by the specification.

Definition 5.3 (Consistency). *Let* Spec *be a specification over an action signature* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*, and* $\hat{t} = (t, a)$ *an annotated test case for* Spec*. Then,* $\hat{t}$ *is* consistent *for* Spec *with respect to* $\sqsubseteq_{\mathtt{ioco}}$ *if it is sound, and for every trace* $\sigma \in ctraces(t)$ *it holds that* $a(\sigma) = pass$ *implies that* σ *is indeed allowed by the specification, i.e.,*

$$\forall\, \sigma \in ctraces(t)\ .\ a(\sigma) = pass \implies \sigma \in traces(\mathsf{Spec})$$

An annotated test suite is consistent with respect to $\sqsubseteq_{\mathtt{ioco}}$ *if all its test cases are.*

As soundness requires that $\sigma \in traces(\mathsf{Spec})$ implies $a(\sigma) = pass$ for every $\sigma \in ctraces(t)$, and consistency additionally requires that $a(\sigma) = pass$ implies $\sigma \in traces(\mathsf{Spec})$, together they require

$$\forall\, \sigma \in ctraces(t)\ .\ a(\sigma) = pass \Longleftrightarrow \sigma \in traces(\mathsf{Spec})$$

Clearly, if a consistent (and hence sound) test suite contains all traces of the specification, it is complete.

Example 5.3. In Example 5.2, the test case t_2 is not consistent, since it allows quiescence in the initial state. The specification does not allow this behaviour, though. Test case t_3 is sound, complete *and* consistent. □

Besides being sound and possibly complete, the test cases annotated according to $a^{\mathtt{ioco}}$ are also consistent. Hence, whenever they detect behaviour that could not occur in any correct implementation, they assign a *fail* verdict. This follows directly from Remark 5.1 and the definition of consistency.

Proposition 5.3. *Let* Spec *be a specification, then the annotated test suite* $\widehat{T} = \{(t, a^{\mathtt{ioco}}_{\mathsf{Spec},t}) \mid t \in \mathcal{T}(\mathsf{Spec})\}$ *is consistent for* Spec *with respect to* $\sqsubseteq_{\mathtt{ioco}}$*.*

Obviously, for all practical purposes test suites definitely should be sound, and preferably complete (although the latter can never be achieved for any non-trivial specification 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 4.1, not the whole possible range of traces that Impl may exhibit will in general be observed during a single test execution. Moreover, although our fairness assumption implies that all traces of Impl will eventually be seen, many executions may be necessary to indeed detect all erroneous behaviour.

5.1 Optimisation: fail-fast and input-minimal tests

The tests from Tretmans' `ioco`-theory [8] 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 5.4 (Optimisations). *Let* Spec *be a specification over an action signature* $(L^{\mathrm{I}}, L^{\mathrm{O}}_{\delta})$*, then*

- *a test t is* fail-fast *with respect to* Spec *if* $\sigma \notin \mathit{traces}(\mathsf{Spec})$ *implies that* $\forall\, a \in L \,.\, \sigma a \notin t$*;*
- *a test t is* input-minimal *with respect to* Spec *if for all* $\sigma a? \in t$ *with* $a? \in L^{\mathrm{I}}$ *it holds that* $\sigma \in \mathit{traces}(\mathsf{Spec})$ *implies* $\sigma a? \in \mathit{traces}(\mathsf{Spec})$*.*

The reason for restricting to fail-fast test cases is that `ioco` defines implementations to be nonconforming if they have at least one nonconforming trace. Hence, once such a trace is observed, the verdict can be given and no further testing is needed. The reason for restricting to input-minimal test cases is that `ioco` allows any behaviour after a trace $\sigma \notin \mathit{traces}(\mathsf{Spec})$ anyway, invalidating the need to test for such behaviour. We note that, in our context of input-enabled specifications, all tests are input-minimal.

Note that for a test case t that is both fail-fast and input-minimal $\sigma a? \in t$ implies $\sigma a? \in \mathit{traces}(\mathsf{Spec})$.

6 Algorithms for test case derivation

So far, we defined a framework in which specifications can be modelled as QIOTSs 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` 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. We describe two algorithms for test case generation: batch testing, or offline testing, that generates a set of test cases first, and then executes these; and on-the-fly or online test case generation, which generates test inputs while executing the system-under-test.

6.1 Batch test case derivation

Algorithm 1 describes batchGen, which generates a set of test cases. The input of this function is a specification Spec and a history $\sigma \in \mathit{traces}(\mathsf{Spec})$. 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, obtaining a test case that can be applied without any start-up phase.

For each call to batchGen, a nondeterministic choice is made. Either the empty test case is returned (used for termination), or a test case is generated that starts by observation, or a test case is generated that starts by stimulation. The fair execution of these alternatives will guarantee eventually the selection of the first alternative, and with that, termination.

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 $\sigma a?$,

Algorithm 1: Batch test case generation for `ioco`.

```
Input: A specification $\mathsf{Spec}$ and history $\sigma \in \mathit{traces}(\mathsf{Spec})$
Output: A test case $t$ for $\mathsf{Spec}$ such that $t$ is input-minimal and fail-fast

procedure batchGen($\mathsf{Spec}, \sigma$)
1    [true] $\to$
2        return $\{\epsilon\}$
3    [true] $\to$
4        result := $\{\epsilon\}$
5        forall $b! \in L^{\mathrm{O}}_{\delta}$ do
6            if $\sigma b! \in \mathit{traces}(\mathsf{Spec})$ then
7                result := result $\cup \{b!\sigma' \mid \sigma' \in \text{batchGen}(\mathsf{Spec}, \sigma b!)\}$
             else
8                result := result $\cup \{b!\}$
             end
         end
9        return result
10   [$a? \in L^{\mathrm{I}}$] $\to$
11       result := $\{\epsilon\} \cup \{a?\sigma' \mid \sigma' \in \text{batchGen}(\mathsf{Spec}, \sigma a?)\}$
12       forall $b! \in L^{\mathrm{O}}$ do
13           if $\sigma b! \in \mathit{traces}(\mathsf{Spec})$ then
14               result := result $\cup \{b!\sigma' \mid \sigma' \in \text{batchGen}(\mathsf{Spec}, \sigma b!)\}$
             else
15               result := result $\cup \{b!\}$
             end
         end
16       return result
```

and, for every possible output action $b! \in L^{\mathrm{O}}$ (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 $\sigma b!$. No traces of the form $b!\sigma'$ (with $\sigma' \neq \epsilon$) are added when the output $b!$ is erroneous; this makes sure that the resulting test case will be fail-fast.

If 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 $b! \in L^{\mathrm{O}}_{\delta}$, some traces of the form $b!\sigma'$, where σ' is a trace from a test case starting with history $\sigma a?$. Again, we stop instantly after an erroneous output.

Remark 6.1. 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 $(\mathsf{Spec}, \sigma, S')$, the conditions in line 6 and 13 would become $\exists\, s \in S' \,.\, b! \in L_{\mathsf{Spec}}(s)$, the recursive calls in line 7 and 14 would add a third parameter $\mathit{reach}_{\mathsf{Spec}}(S', b!)$, and the recursive call in line 11 would add a third parameter $\mathit{reach}_{\mathsf{Spec}}(S', a?)$. □

Remark 6.2. Clearly, it is impossible to explicitly store any nontrivial test case for a specification over an infinite number of output 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. $\square$

Theorem 6.1. *Let* Spec *be a specification, and* $t = \mathrm{batchGen}(\mathsf{Spec}, \epsilon)$*. Then,* t *is a fail-fast and input-minimal test case for* Spec*.*

Proof (sketch). We need to show that the conditions of Definition 4.1 are satisfied. Prefix-closedness can be shown using induction over the length of a trace, as every step of the algorithm suffixes at most one action and also returns ϵ. Furthermore, as every iteration of the algorithm increases the length of the test case by one, a test case obtained by running the algorithm (a finite amount of time) can never have an infinite increasing sequence. Finally, the required structure of the traces precisely corresponds to what's happening in the three nondeterministic choices: either suffixing nothing, suffixing all outputs including δ, or suffixing an input and all output actions excluding δ. $\square$

Note that Propositions 5.1 and 5.3 imply that $\hat{t} = (t, a^{\mathtt{ioco}}_{\mathsf{Spec},t})$ is sound and consistent for Spec with respect to $\sqsubseteq_{\mathtt{ioco}}$.

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

Theorem 6.2. *Let* Spec *be a specification, and* T *the set of all linear test cases that can be generated using Algorithm 1. Then, the annotated test suite* $\widehat{T} = \{(t, a^{\mathtt{ioco}}_{\mathsf{Spec},t}) \mid t \in T\}$ *is complete for* Spec *with respect to* $\sqsubseteq_{\mathtt{ioco}}$*.*

Proof. By Proposition 5.2 we know that $\widehat{T}$ is complete for Spec with respect to $\sqsubseteq_{\mathtt{ioco}}$ if for all $\sigma \in \mathit{canon}(\mathit{traces}(\mathsf{Spec}))$ either the specification allows all outputs (including quiescence) after σ, or there exists an annotated test case $(t, a) \in \widehat{T}$ such that $\sigma\delta \in t$.

Let $\sigma = a_1 a_2 \dots a_n \in \mathit{canon}(\mathit{traces}(\mathsf{Spec}))$. 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 $a?$ 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 $a_1 \in L^{\mathrm{O}}_{\delta}$, then we choose to observe. This results again in several recursive calls; one for every $b! \in L^{\mathrm{O}}_{\delta}$. 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. $\square$

Clearly, this implies that the set of all (not necessarily linear) test cases that can be generated using Algorithm 1 is complete. Still, some issues need to be taken into consideration.

First, 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, although the set of all test cases derivable using the algorithm is in theory complete, this does not necessarily mean that every erroneous implementation is detected by running all of these tests once. After all, because of nondeterminism, faulty behaviour might not show during testing, even though it may turn up afterwards. Only if all possible outcomes of every nondeterministic choice are guaranteed to be taken at least once during testing, a complete test suite can indeed observe all possible erroneous traces of an implementation. We refer to [13] for more details on expected test coverage (including probabilistic computations).

Despite these restrictions, the completeness theorem provides 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.

6.2 On-the-fly test case derivation

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 in Algorithm 2. We note that the efficiency considerations of Remark 6.1 also apply to this algorithm.

The input of the algorithm consists of a specification Spec and a concrete implementation Impl. The algorithm contains one local variable, σ, which represents the trace obtained thus far; it is therefore initialised with the empty trace ϵ. Then, the while loop is executed a nondeterministic number of times.

For every test step there is a nondeterministic choice between ending the test, observing, or stimulating the implementation by any of the input actions. 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 $\mathit{traces}(\mathsf{Spec})$. 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. 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

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

Input: A specification Spec, a concrete implementation Impl.
Output: The verdict *pass* when the observed behaviour of Impl was `ioco`-conform Spec, and the verdict *fail* when a nonconforming trace was observed during the test.

```
1  $\sigma := \epsilon$
2  while true do
3      [true] $\rightarrow$
4              return pass
5      [true] $\rightarrow$
6              observe Impl's next output $b!$ (possibly $\delta$)
7              $\sigma := \sigma b!$
8              if $\sigma \notin traces(\mathsf{Spec})$ then return fail
9      [$a? \in L^{\mathrm{I}}$] $\rightarrow$
10             try
11                 atomic
12                     stimulate Impl with $a?$
13                     $\sigma := \sigma a?$
                   end
14             catch an output $b!$ occurs before $a?$ could be provided
15                 $\sigma := \sigma b!$
16                 if $\sigma \notin traces(\mathsf{Spec})$ then return fail
               end
   end
17 return pass
```

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 6.3. *Algorithm 2 is sound and consistent with respect to* $\sqsubseteq_{\texttt{ioco}}$.

Proof. We first prove soundness. Note that σ keeps track of the trace exhibited by the implementation thus far. The only way for the algorithm to return *fail* is when $\sigma \notin traces(\mathsf{Spec})$ after an observation. In that case, indeed we found that $traces(\mathsf{Impl}) \not\subseteq traces(\mathsf{Spec})$ and hence $\mathsf{Impl} \not\sqsubseteq_{\texttt{ioco}} \mathsf{Spec}$.

For consistency, note that the only way for the algorithm to return *pass* is when $\sigma \in traces(\mathsf{Spec})$ by the end of the last iteration. As the on-the-fly algorithm basically is a test case with only one complete trace, this directly satisfies the definition of consistency. □

The algorithm is obviously not complete when run only once. However, it is easy to see that, just like for the batch test case generation algorithm, there is no erroneous implementation that cannot be caught in principle. The more often it is run, the more likely that erroneous transitions are detected.

7 Conclusions and future work

This paper has revisited the `ioco`-theory for model-based testing so that it can handle divergences, i.e., τ-loops. Divergences are common in practice, for instance as a result of action hiding. Hence, our results extend model-based testing techniques to an important class of new models.

We have phrased `ioco`-theory in a trace-based setting, using only standard concepts from labelled transition systems. Technically, our treatment of divergence proceeds via the QIOTS model, where quiescence is modelled as a special output action. QIOTSs constitute a clean modelling framework, closed under parallel composition, action hiding and determinization. This paves the way to further study compositionality results; compositionality is widely recognized as one of the most crucial techniques to handle the complexity of today's systems. Further, testers can be oblivious of the QIOTS model, since any input/output transition system can be transformed into a QIOTS via a deltafication operator.

This work spawns two directions for future research. First, our setting requires that τ-loops contain finitely many states only. This restriction is needed to ensure well-formedness of the deltification operator. Second, as mentioned, it is interesting to study compositionality results for systems with divergences.

Acknowledgements We would like to thank the reviewers for their thorough comments that really helped improve this paper. We thank Gerjan Stokkink for his large contributions to work that provides important ingredients for this paper [2, 14–16].

Since this paper is a part of the Festschrift at the occasion of the 60th birthday of Kim Guldstrand Larsen, we like to thank Kim for the many exciting and fruitful discussions we have had, and still have, over all these years in project meetings, at conferences and many other occasions — Kim was never quiescent.

References

1. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software — Concepts and Tools **17**(3) (1996) 103–120
2. Stokkink, W.G.J., Timmer, M., Stoelinga, M.: Divergent quiescent transition systems. In: Proceedings of the 7th International Conference on Tests and Proofs (TAP). Volume 7942 of Lecture Notes in Computer Science., Springer (2013) 214–231
3. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly **2** (1989) 219–246
4. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
5. Timmer, M., Brinksma, E., Stoelinga, M.I.A.: Model-based testing. In: Software and Systems Safety: Specification and Verification. Volume 30 of NATO Science for Peace and Security Series D. IOS Press, Amsterdam (2011) 1–32
6. Vaandrager, F.W.: On the relationship between process algebra and input/output automata (extended abstract). In: Proceedings of the 6th Annual Symposium on Logic in Computer Science (LICS), IEEE Computer Society (1991) 387–398

7. Tretmans, J.: Test generation with inputs, outputs, and quiescence. In: Proceedings of the 2nd International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS). Volume 1055 of Lecture Notes in Computer Science., Springer (1996) 127–146
8. Tretmans, J.: Model based testing with labelled transition systems. In: Formal Methods and Testing. Volume 4949 of Lecture Notes in Computer Science., Springer (2008) 1–38
9. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer **7**(4) (2005) 297–315
10. Lynch, N.A., Tuttle, M.R.: Hierarchical correctness proofs for distributed algorithms. In: Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC), ACM (1987) 137–151
11. De Nicola, R., Segala, R.: A process algebraic view of input/output automata. Theoretical Computer Science **138** (1995) 391–423
12. Tarjan, R.E.: Depth-first search and linear graph algorithms (working paper). In: Proceedings of the 12th Annual Symposium on Switching and Automata Theory (SWAT), IEEE Computer Society (1971) 114–121
13. Stoelinga, M., Timmer, M.: Interpreting a successful testing process: Risk and actual coverage. In: Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE Computer Society (2009) 251–258
14. Stokkink, W.G.J., Timmer, M., Stoelinga, M.I.A.: Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation. In: Proceedings of the 7th Workshop on Model-Based Testing (MBT). Volume 80 of EPTCS. (2012) 73–87
15. Stokkink, W.G.J., Timmer, M., Stoelinga, M.I.A.: Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation (extended version). Technical Report TR-CTIT-12-05, University of Twente (2012)
16. Stokkink, G.: Quiescent transition systems. Master's thesis, University of Twente (2012)