Matching Implementations to Specifications: The Corner Cases of ioco

Ramon Janssen
Radboud University, Nijmegen
ramonjanssen@cs.ru.nl

Jan Tretmans
TNO-ESI, Eindhoven
Radboud University, Nijmegen
tretmans@cs.ru.nl

ABSTRACT

A well-known conformance relation for model-based testing is ioco. A conformance relation expresses when an implementation is correct with respect to a specification. Unlike many other conformance and refinement relations, ioco has different domains for implementations and for specifications. Consequently, ioco is neither reflexive nor transitive, implying that a specification does not implement itself, and that specifications cannot be compared for refinement. In this paper, we investigate how we can compensate for the lack of reflexivity and transitivity. We show that (*i*) given a specification, we can construct in a standard way a canonical conforming implementation that is very 'close' to the specification; and (*ii*) a refinement preorder on specification models can be defined such that a refined model allows less ioco-conforming implementations. We give declarative and constructive definitions of both, we give examples of unimplementable corner-cases, we investigate decidability, and we do that for ioco as well as for the ioco-variant uioco. The latter turns out to be simpler and on more aspects decidable.

ACM Reference Format:
Ramon Janssen and Jan Tretmans. 2019. Matching Implementations to Specifications:, The Corner Cases of ioco. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3297280.3297496

1 INTRODUCTION

Software testing involves checking of desired properties of a software product by systematically executing the software, while stimulating it with inputs, and observing and checking outputs. *Model-Based Testing* (MBT) is a form of black-box testing where a System Under Test (SUT) is tested for conformance to a model. The model specifies, in a formal way, what the system is allowed to do and what it shall not do. As such, the model is the basis for the algorithmic generation of test cases and for the evaluation of test results.

An important prerequisite for MBT is the precise definition of what it means for an SUT to conform to its model. Conformance is expressed using an *implementation relation* or *conformance relation.* Although an SUT is a black box, we can assume it could be modelled by some model instance in a domain of implementation models. This assumption is commonly referred to as the *testability hypothesis*, or *testability assumption* [7]. This assumption allows reasoning about SUTs as if they were formal models, and makes it possible to define a conformance relation as a formal relation between the domain of specification models and the domain of implementation models.

One of the formal theories for model-based testing uses Labelled Transition Systems (LTSes) as models and *ioco* (**i**nput-**o**utput-**co**nformance) as conformance relation [17, 18]. An LTS is a structure with states, representing the states of the actual system, and with transitions between states representing the actions that the system may perform. Actions can be inputs, outputs, or internal steps. The conformance relation ioco expresses that an SUT conforms to its specification if the SUT never produces an output that cannot be produced by the specification in the same situation. A particular, virtual output is *quiescence*, actually expressing the absence of real outputs. The ioco-testing theory for LTSes provides a test generation algorithm that is *sound* and *exhaustive*, i.e., the (possibly infinitely many) test cases generated from an LTS model detect all and only ioco-incorrect implementations. The ioco-testing theory constitutes a well-defined theory of model-based testing, and it forms the basis for various practical MBT tools, like TorX, TGV, Uppaal-Tron, Axini Test Manager, JTorx, and TorXakis.

Many conformance relations from the literature have nice properties, such as being reflexive and transitive on the domain of models (i.e., they are preorders). Reflexivity implies that any model is a correct implementation of itself, and transitivity enables step-wise refinement: a high-level specification is refined to more detailed design models, which are refined to an implementation model. Hence, such a preorder is often called a *refinement relation.* Examples are trace inclusion, failure preorder [9], and alternating refinement [1].

The ioco conformance relation, however, is neither reflexive nor transitive, the reason being that the domains of specifications and implementations differ. An implementation is assumed to be an LTS that is *input-enabled*, that is, any input to the implementation is accepted in every state, whereas specifications are just LTSes. Inputs that are not accepted in a specification state are considered to be *underspecified*: no behaviour is specified for such inputs, thus leaving implementation freedom to the implementer.

In this paper, we will investigate *quasi*-reflexivity and *quasi*-transitivity for ioco. Specifically, the following questions arise:

($Q1$) Given a specification, is it possible to construct in a standard way an implementation that is akin to the specification, i.e., can we derive a canonical conforming implementation from a specification?

We thank Frits Vaandrager for his valuable feedback.

SAC '19, April 8–12, 2019, Limassol, Cyprus
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297496

($Q2$) Is it possible to define an ioco-refinement preorder on the domain of specification models, such that a refined model allows less ioco-conforming implementations?

Question ($Q1$) addresses a practical use of specifications: they should be effectively ioco-implementable, preferably in an algorithmic manner. In ioco-literature, ($Q1$) is often approached by *angelic completion*: a non-input enabled specification is turned into an input-enabled one, by adding an input self-loop to each state where that input is not accepted. We will discuss this approach in Section 4. We will also re-discuss the common informal interpretation of such not-accepted inputs in specifications, i.e., that "an implementation is completely free to do anything it likes after [that input]" [18], and we will show that this interpretation is at least inaccurate.

Question ($Q2$) naturally leads to an equivalence on specifications by mutual refinement, expressing preservation of ioco-conformance. More generally, the answer to ($Q2$) leads to a specification lattice with the usual lattice-operations such as join (disjunction of specifications), meet (conjunction of specifications), top element (universal specification), and bottom element (unimplementable specification).

The answers to ($Q1$) and ($Q2$) are first investigated via a couple of corner-case examples in Section 3, and then elaborated in Section 4 based on so-called *conformal traces* and an *ioco-characterization* of specifications based on these traces. But these first answers will only address the "is it possible"-part of the questions; conformal traces are not at all constructive and therefore not easily computable. Therefore, in two steps, we will turn this into a more constructive approach. First, in Section 5, we define a class of languages with quiescence, named *suspension languages*, for which a canonical implementation can be derived. Secondly, in Section 6, we introduce a construction on LTSes to obtain the ioco-characterization of a specification, thus leading to an algorithmic answer to ($Q1$) and ($Q2$) for finite specifications. Correctness follows from the language-theoretic results in Section 5. For infinite specifications, we prove undecidability of the ioco-characterization. Lastly, in Section 7, we consider the conformance relation *uioco* which is a slight variation of ioco [4]. We show that a similar characterization for uioco is simpler than the ioco-characterization, and moreover decidable.

Summarizing, our main contributions are

- a formal characterization of specifications, for ioco and uioco,
- a translation between suspension languages and implementation models,
- a structured approach for answering ($Q1$) and ($Q2$), and
- decidability results for the characterizations.

Proofs can be found in the technical report [10].

1.1 Related Work

As the introduced questions are so natural, solutions have been proposed, but mostly partial ones. Our work is thus strongly inspired by these solutions, which we combine into a coherent theory.

Most notably, Bourdonov and Kossatchev [5] remark that some ioco-specifications contain traces, not contained in any conforming implementation, which they name *nonconformal*. They show that a sequence of transformations can remove these traces, to obtain a reflexive extension of ioco. We base our characterization upon these traces, and show that the resulting relation can be generalized to a preorder of language inclusion to answer questions ($Q1$) and ($Q2$).

Willemse [20] identified constraints on trace sets which capture precisely the traces of ioco-specifications. This yields a trace characterization of a different form, suitable for reasoning about the correspondence between LTSes and languages. Beneš et al. [2] detect *invalid* specifications which violate these constraints, resulting from compositions of initially valid specifications. They also introduce transformations from invalid specifications to valid ones. We show that the characterization of nonconformal traces is strongly related to the one by Willemse, and that the transformations of Beneš et al. can be used to remove nonconformal traces.

Volpato and Tretmans [19] investigate question ($Q2$) for the conformance relation uioco, instead of ioco. They analyse dependencies between traces of specifications, similarly to Willemse, in order to reduce redundant test cases. The analysis is limited to trace sets of specifications, and as these are generally infinite, no constructive or algorithmic approaches for LTSes follow. Furthermore, they claim that uioco is to be preferred over ioco, but no explicit motivation is given. Our work thus improves on this by giving constructive characterizations, and by comparing the two conformance relations.

2 PRELIMINARIES

We first recall the basics of labelled transition systems and ioco theory. We refer to [18] for a more elaborate overview.

2.1 Labelled Transition Systems

Definition 2.1. A *labelled transition system* (LTS) with inputs and outputs is a 5-tuple (Q, A_I, A_U, T, q^0), where

- Q is a non-empty, countable set of states,
- A_I and A_U are disjoint sets of input and output actions,
- $T \subseteq Q \times (A_I \cup A_U \cup \{\tau\}) \times Q$ is a transition relation, and
- $q^0 \in Q$ is the initial state.

The special action $\tau \notin A_I \cup A_U$ denotes the occurrence of an unobservable, internal transition. We make the usual assumption that there are no infinite sequences $(q, \tau, q'), (q', \tau, q''), \ldots$ of τ-transitions. The domain of LTSes with this assumption is denoted $\mathcal{LTS}(A_I, A_U)$, for inputs A_I and outputs A_U. Given an LTS s, we write Q_s, T_s and q_s^0 for respectively its states, transitions and initial state. We fix A_I and A_U as disjoint sets of inputs and outputs with $A = A_I \cup A_U$, unless stated otherwise, for the remainder of this paper. We use $\mathcal{LTS}$ as a shorthand for $\mathcal{LTS}(A_I, A_U)$. This is our domain of specifications.

Standard notation is used to express sets of traces as for formal languages: $\sigma_1 \cdot \sigma_2$ or $\sigma_1\sigma_2$ denotes concatenation; σ^n denotes repetition; σ^* denotes the Kleene-star, with $\sigma^+ = \sigma\sigma^*$; and ϵ denotes the empty sequence. We use the following auxiliary definitions.

Definition 2.2. Let $s \in \mathcal{LTS}$; $q, q' \in Q_s$; $Q \subseteq Q_s$; $\ell \in A$; $\sigma \in A^*$; $\ell_\tau \in A \cup \{\tau\}$ and $\sigma_\tau \in (A \cup \{\tau\})^*$. Then we define

$$\begin{array}{llr}
q \xrightarrow{\epsilon} q' & \overset{\text{def}}{\Leftrightarrow} q = q' & (\rightarrow\, \in Q \times (A \cup \{\tau\})^* \times Q) \\
q \xrightarrow{\sigma_\tau \ell_\tau} q' & \overset{\text{def}}{\Leftrightarrow} \exists q'' \in Q_s : q \xrightarrow{\sigma_\tau} q'' \wedge (q'', \ell_\tau, q') \in T_s & \\
q \overset{\epsilon}{\Rightarrow} q' & \overset{\text{def}}{\Leftrightarrow} \exists n \in \mathbb{N} : q \xrightarrow{\tau^n} q' & (\Rightarrow\, \in Q \times A^* \times Q) \\
q \overset{\sigma\ell}{\Longrightarrow} q' & \overset{\text{def}}{\Leftrightarrow} \exists q_1, q_2 \in Q_s : q \overset{\sigma}{\Rightarrow} q_1 \xrightarrow{\ell} q_2 \overset{\epsilon}{\Rightarrow} q' & \\
q \overset{\sigma}{\Rightarrow} & \overset{\text{def}}{\Leftrightarrow} \exists q' : q \overset{\sigma}{\Rightarrow} q' & (\text{we overload} \Rightarrow\, \in Q \times A^*)
\end{array}$$

$$\begin{array}{llll}
\text{traces}\,(q) & \overset{\text{def}}{=} \{\sigma \in A^* \mid q \overset{\sigma}{\Rightarrow}\} & q \text{ after } \sigma & \overset{\text{def}}{=} \{q' \in Q_s \mid q \overset{\sigma}{\Rightarrow} q'\} \\
\text{traces}\,(s) & \overset{\text{def}}{=} \text{traces}\,(q_s^0) & s \text{ after } \sigma & \overset{\text{def}}{=} \{q_s^0 \text{ after } \sigma\}
\end{array}$$

$$\begin{aligned}
&\text{traces}\,(Q) \overset{\text{def}}{=} \bigcup\{\text{traces}\,(q) \mid q \in Q\}\\
&Q \text{ after } \sigma \overset{\text{def}}{=} \bigcup\{q \text{ after } \sigma \mid q \in Q\}\\
&\text{init}(Q) \overset{\text{def}}{=} \{\ell \in A \mid \exists q \in Q : q \xrightarrow{\ell}\} \qquad \text{out}(Q) \overset{\text{def}}{=} \text{init}(Q) \cap A_U\\
&s \text{ is } \textit{input-enabled} \overset{\text{def}}{\Leftrightarrow} \forall q \in Q_s : \forall a \in A_I : q \overset{a}{\Rightarrow}\\
&s \text{ is } \textit{deterministic} \overset{\text{def}}{\Leftrightarrow} \forall q \in Q_s : \forall \sigma \in \text{traces}\,(q) : |q \text{ after } \sigma| = 1\\
&\mathcal{IOTS} \overset{\text{def}}{=} \{s \in \mathcal{LTS} \mid s \text{ is input-enabled}\}
\end{aligned}$$

To avoid ambiguity, we sometimes add the name of an LTS as a subscript to the introduced notation, e.g. we distinguish $q \text{ after}_s\, \sigma$ and $q \text{ after}_t\, \sigma$, if $q \in Q_s$ and $q \in Q_t$. Our domain of implementations is $\mathcal{IOTS}$, so our testability assumption is that a system under test behaves as if it were an IOTS model.

2.2 Quiescence and ioco

An environment can supply an IOTS with inputs from A_I, and observe outputs in A_U. When a system is in a state without any output or internal transitions, it cannot change state by itself, and it will stay *quiescent*. Quiescence can be observed by the environment, in practice by waiting for an output until a time-out has expired. This is made explicit by adding self-loops with virtual output action $\delta \notin A \cup \{\tau\}$, representing quiescence. We fix the sets of actions $A_U^\delta = A_U \cup \{\delta\}$ and $A^\delta = A \cup \{\delta\}$. The notation of Definition 2.2 is also used with δ as an output.

Definition 2.3. Let $s \in \mathcal{LTS}(A_I, A_U)$, and $q \in Q_s$. Then

$$q \text{ is } \textit{quiescent} \text{ in } s \overset{\text{def}}{\Leftrightarrow} \neg\exists \ell \in A_U \cup \{\tau\} : q \xrightarrow{\ell}$$

The *deltafication* [16] of s is defined as $\Delta(s) \in \mathcal{LTS}(A_I, A_U^\delta)$, with

$$\begin{aligned}
&Q_{\Delta(s)} \overset{\text{def}}{=} Q_s\\
&T_{\Delta(s)} \overset{\text{def}}{=} T_s \cup \{(q, \delta, q) \mid q \in Q_s, q \text{ is quiescent in } s\}\\
&q^0_{\Delta(s)} \overset{\text{def}}{=} q^0_s
\end{aligned}$$

The *suspension traces* of s are defined as $\text{Straces}\,(s) \overset{\text{def}}{=} \text{traces}\,(\Delta(s))$

Conformance of implementations to specifications is expressed with the ioco relation [17], based on the deltafication. Intuitively, the implementation should only produce outputs appearing in the specification, including quiescence, after specified suspension traces. Absence of an input in specifications denotes underspecification: any behaviour is allowed afterwards.

Definition 2.4. Let $i \in \mathcal{IOTS}, s \in \mathcal{LTS}$. Then

$$i \textbf{ ioco } s \overset{\text{def}}{\Leftrightarrow} \forall \sigma \in \text{Straces}\,(s) : \text{out}(\Delta(i) \text{ after } \sigma) \subseteq \text{out}(\Delta(s) \text{ after } \sigma)$$

Since $\mathcal{IOTS} \subseteq \mathcal{LTS}$, every implementation can also act as a specification. When only relating implementations, ioco is already a preorder, namely that of suspension trace inclusion. Furthermore, a weak form of transitivity holds.

Lemma 2.5. *[18] Let $i, i' \in \mathcal{IOTS}$, $s \in \mathcal{LTS}$. Then*

$$i \textbf{ ioco } i' \iff \text{Straces}\,(i) \subseteq \text{Straces}\,(i')$$
$$i \textbf{ ioco } i' \wedge i' \textbf{ ioco } s \implies i \textbf{ ioco } s \qquad \textit{(quasi-transitivity)}$$

2.3 Refinement

To address question ($Q2$) posed in the introduction, we define an explicit notion of refinement for ioco: does one specification have less conforming implementations than another? Likewise, equivalence expresses having the same implementations.

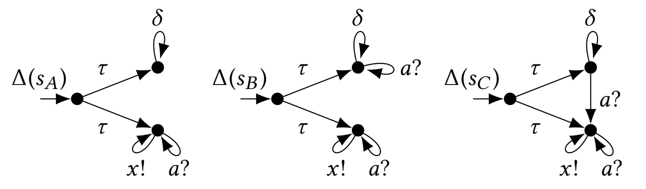

Figure 1: LTSes s_A, s_B and s_C, with $A_I = \{a?\}$ and $A_U = \{x!\}$. Deltafication has been applied.

Definition 2.6. Let $s, s' \in \mathcal{LTS}$. Then

$$\begin{aligned}
s \preccurlyeq s' &\overset{\text{def}}{\Leftrightarrow} \forall i \in \mathcal{IOTS} : i \textbf{ ioco } s \implies i \textbf{ ioco } s'\\
s \simeq s' &\overset{\text{def}}{\Leftrightarrow} s \preccurlyeq s' \wedge s' \preccurlyeq s
\end{aligned}$$

3 CORNER-CASE EXAMPLES

In the introduction we mentioned some commonly used approaches and interpretations for ioco, such as making a specification model input-enabled by angelic completion, implementation freedom for underspecified inputs, and implementability of specifications. In this section we show three examples that illustrate that these approaches and interpretations are not completely accurate, which is relevant when answering ($Q1$) and ($Q2$).

Example 3.1 (Angelic completion). Consider specification s_A in Figure 1. Specification s_A is not input-enabled: the upper-right state does not have an $a?$ transition. Applying angelic completion, i.e., adding an $a?$-self-loop in the upper-right state, results in s_B. But s_B is not ioco-conforming to s_A, since $\text{out}(\Delta(s_B) \text{ after } a?) = \{\delta, x!\} \nsubseteq \text{out}(\Delta(s_A) \text{ after } a?) = \{x!\}$. So, adding self-loops to non-input-enabled specification states may result in non-ioco conforming implementations. This shows that missing inputs are not underspecified *per state*, but only *per trace*: trace $a?$ is specified (by the lower branch), so performing angelic completion (which modifies the behaviour after $a?$ in the upper branch) does not result in a conforming implementation.

Adding the missing $a?$-transition in the way it is done in s_C, does result in a conforming implementation: s_C **ioco** s_A.

Example 3.2 (Implementation freedom). The only difference between s_A and s_C is that s_A has an underspecified input-transition, whereas s_C does not. Because of this, $\delta a? \in \text{Straces}\,(s_C)$, whereas $\delta a? \notin \text{Straces}\,(s_A)$. Since unspecified traces would lead to implementation freedom, one would expect that s_A is a more liberal specification than s_C, and that it would allow more conforming implementations.

This, however, is not the case: since δ-transitions are always self-loops, any potential implementation i having suspension trace $\delta a? \delta$, will also have $a? \delta$. And having trace $a? \delta$ implies that $\delta \in \text{out}(i \text{ after } a?)$, whereas $\delta \notin \text{out}(s_A \text{ after } a?)$. Consequently, any conforming implementation i **ioco** s_A cannot have the suspension trace $\delta a? \delta$, and thus, despite the seemingly underspecified trace $\delta a?$, there is no full implementation freedom after $\delta a?$.

Actually, we will later show that $s_A \simeq s_C$, so they allow the same implementations: s_C explicitly specifies the behaviour after $\delta a?$, whereas s_A does so implicitly, through the dependency between traces such as $\delta a? \delta$ and $a? \delta$.

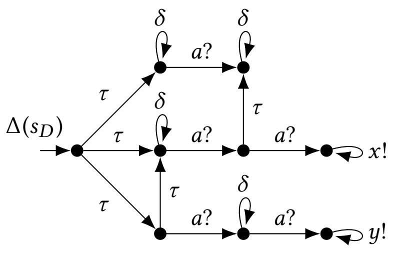

Figure 2: Specification s_D, with $A_I = \{a?\}$ and $A_U = \{x!, y!\}$.

Example 3.3 (Implementability). Consider s_D in Figure 2, introduced by Bourdonov and Kossatchev [5]. Assume an implementation $i \in \mathcal{IOTS}$ with i **ioco** s_D. We have that $\delta a?\delta a? \in$ Straces (i), because:

(*1*) $\epsilon \in$ Straces (i); (*2*) then out($\Delta(i)$ after ϵ) $\neq \emptyset$; moreover, out($\Delta(s_D)$ after ϵ) $= \{\delta\}$, so it must be that out($\Delta(i)$ after ϵ) $= \{\delta\}$, and consequently $\delta \in$ Straces (i); (*3*) then also $\delta a? \in$ Straces (i) since i is input-enabled; (*4*) analogous to (*2*): out($\Delta(s_D)$ after $\delta a?$) $= \{\delta\}$, thus out($\Delta(i)$ after $\delta a?$) $= \{\delta\}$, so also $\delta a?\delta \in$ Straces (i); (*5*) analogous to (*3*): $\delta a?\delta a? \in$ Straces (i), as i is input-enabled.

Let out(i after $\delta a?\delta a?$) $= X$, then, because $\delta a?\delta a? \in$ Straces (i) it holds that $X \neq \emptyset$ (since there is either a 'real' output $x!$ or $y!$, and if not, there is output δ). Moreover, since δ-transitions are always added as loops in $\Delta(i)$, we can leave them out, and the resulting traces will at least have the same outputs as after $\delta a?\delta a?$:

$$\text{out}(\Delta(i) \text{ after } \delta a?a?) \supseteq X \quad \text{and} \quad \text{out}(\Delta(i) \text{ after } a?\delta a?) \supseteq X$$

Furthermore, if i **ioco** s_D holds, then we must have that: out($\Delta(i)$ after $\delta a?a?$) $\subseteq$ out($\Delta(s_D)$ after $\delta a?a?$) $= \{x!\}$, and, likewise, out($\Delta(i)$ after $a?\delta a?$) $\subseteq$ out($\Delta(s_D)$ after $a?\delta a?$) $= \{y!\}$.

Combining all these constraints for X, we conclude that there is no possible X satisfying all of them. This implies that the conforming implementation i cannot exist: s_D is a specification that has no conforming implementations at all. Apparently, there exist *unimplementable* specifications. Of course, this might pose a problem when considering some form of reflexivity.

4 TRACE CHARACTERIZATION OF IOCO

As a first step towards formalizing the examples in the previous section, and partly answering questions ($Q1$) and ($Q2$), we characterize every specification by a set of traces.

This characterization is motivated by the fact that a specification shall specify which behaviour of implementations is allowed and which behaviour is forbidden. Behaviour of implementations is expressed as suspension traces, so a specification describes which suspension traces are allowed, and which are not. This is most easily done if specifications are also characterized explicitly by a set of suspension traces. This set, however, is not the same as the suspension traces of the specification as defined in Definition 2.3. Sometimes the characterization set is larger, if underspecified traces are allowed, and sometimes it is smaller, if the set of suspension traces according to Definition 2.3 contains traces which cannot occur in any conforming implementation, as was illustrated in Example 3.3. Following Bourdonov and Kossatchev [5] we call such unimplementable traces nonconformal traces, or conversely, an *ioco-conformal trace* is a suspension trace that can occur in some ioco-conforming implementation.

Definition 4.1. Let $\sigma \in (A^\delta)^*$ and $s \in \mathcal{LTS}$. Then

$$\sigma \textbf{ iocfl } s \overset{\text{def}}{\Leftrightarrow} \exists i \in \mathcal{IOTS} : i \textbf{ ioco } s \land \sigma \in \text{Straces}(i)$$

$$\langle s \rangle_{\textbf{ioco}} \overset{\text{def}}{\Leftrightarrow} \{\sigma \in (A^\delta)^* \mid \sigma \textbf{ iocfl } s\}$$

We call $\langle s \rangle_{\textbf{ioco}}$ the *ioco-characterization* of s.

We now revisit the examples of Section 3, and interpret these examples in terms of conformal traces.

Example 4.2 (Angelic completion). Adding self-loops to s_A, as done in Example 3.1, leads to adding the trace $a?\delta$, which is a nonconformal trace of s_A. No conforming implementation i can contain this trace because, if it would, it would also have: $\delta \in$ out($\Delta(i)$ after $a?$) $\nsubseteq$ out($\Delta(s_A)$ after $a?$) $= \{x!\}$.

Example 4.3 (Implementation freedom). In Example 3.2 we argued that the trace $\delta a?\delta$ is underspecified in s_A, yet, it cannot be implemented in any conforming implementation. This means that it is nonconformal. If an implementation i would implement $\delta a?\delta$, then, because δ-transitions always occur as loops in $\Delta(i)$, also $a?\delta$ would occur, which leads to non-conformance as above.

Actually, our claim in Example 3.2 that $s_A \simeq s_C$, can be proved by showing that $\langle s_A \rangle_{\textbf{ioco}} = \langle s_C \rangle_{\textbf{ioco}}$.

Example 4.4 (Implementability). Specification s_D in Example 3.3 does not have any conforming implementations, so it also has no conformal traces: $\langle s_D \rangle_{\textbf{ioco}} = \emptyset$.

4.1 Properties of ioco Characterizations

Ioco characterizations of specifications and implementations have a couple of nice properties which, together with Lemma 2.5, already partly answer our questions ($Q1$) and ($Q2$).

THEOREM 4.5. *Let $i \in \mathcal{IOTS}$ and $s, s' \in \mathcal{LTS}$. Then*

(*1*) $\langle i \rangle_{\textbf{ioco}} =$ Straces (i)
(*2*) $\langle i \rangle_{\textbf{ioco}} \subseteq \langle s \rangle_{\textbf{ioco}} \iff i \textbf{ ioco } s$
(*3*) $\langle s \rangle_{\textbf{ioco}} \subseteq \langle s' \rangle_{\textbf{ioco}} \iff s \preccurlyeq s'$
(*4*) $\langle s \rangle_{\textbf{ioco}} = \langle s' \rangle_{\textbf{ioco}} \iff s \simeq s'$

Question ($Q1$), construction of a canonical conforming implementation i_s from a specification s, can be answered by taking $i_s \in \mathcal{IOTS}$ such that Straces $(i_s) = \langle s \rangle_{\textbf{ioco}}$. According to Theorem 4.5.1 and 4.5.2 we then have i_s **ioco** s. As Examples 3.3 and 4.4 show, this approach is only possible if s is implementable, that is, if $\langle s \rangle_{\textbf{ioco}} \neq \emptyset$. Question ($Q2$), definition of an ioco-refinement preorder, follows directly from Theorem 4.5.3.

The answers are partial, because, though well-defined, these definitions do not help in actually constructing the canonical implementation nor in checking refinement, since Definition 4.1 is not at all constructive. It is expressed in terms of conformal traces, which in turn are expressed in terms of the existence of a conforming implementation. In the next sections we will give more constructive descriptions of conformal-trace sets and ioco-characterizations. Theorem 4.5.1 shows that this will be relatively easy for input-enabled implementations. For specifications, however, it is more intricate and involves both adding to and removing traces from Straces (s), as the examples in the previous section showed.

The more constructive approach is given in two steps. First, in Section 5, we define *suspension languages* to characterize specifications and implementations. Second, in Section 6, we will show how the manipulation of suspension languages can be lifted to constructive transformations on labelled transition systems.

5 SUSPENSION LANGUAGES

We will now present an alternative formulation of the ioco characterization, expressed as a set of constraints on its traces. These constraints follow from the construction of implementations, as traces of $\Delta(i)$ for some IOTS i. They are inspired by the rules of Willemse [20] for LTSes with explicit quiescence, which capture whether the traces of such an LTS are the suspension traces of any specification in $\mathcal{LTS}$. This characterization leads to a correspondence between implementations and languages with quiescence.

Definition 5.1. A trace $\sigma \in (A^{\delta})^*$ is *anomalous* if it contains an output x following δ, that is, if $\sigma = \sigma_1 \delta x \sigma_2$ for some $\sigma_1, \sigma_2 \in (A^{\delta})^*$ and $x \in A_U$.

Definition 5.2. In the following definitions, let σ and ρ range over $(A^{\delta})^*$ and let ℓ range over A^{δ}. A language $L \subseteq (A^{\delta})^*$ is
– *prefix-closed* $\overset{\text{def}}{\Leftrightarrow} \forall \sigma\ell \in L : \sigma \in L$
– *input-enabled* $\overset{\text{def}}{\Leftrightarrow} \forall \sigma \in L : \forall a \in A_I : \sigma a \in L$,
– *non-blocking* $\overset{\text{def}}{\Leftrightarrow} \forall \sigma \in L : \exists x \in A_U^{\delta} : \sigma x \in L$
– *anomaly free* $\overset{\text{def}}{\Leftrightarrow} \forall \sigma \in L : \sigma$ is not anomalous
– *quiescence reducible* $\overset{\text{def}}{\Leftrightarrow} \forall \sigma\delta\rho \in L : \sigma\rho \in L$
– *quiescence stable* $\overset{\text{def}}{\Leftrightarrow} \forall \sigma\delta\rho \in L : \sigma\delta\delta\rho \in L$
Language L is a *suspension language* if $L \neq \emptyset$ and if all of the above holds. The domain of suspension languages is denoted by $\mathcal{SL}$.

Prefix-closedness and non-emptiness hold for the traces of any LTS. Input-enabledness corresponds to the equally named property on LTSes, and holds by definition for any IOTS. The remaining properties arise from Definition 2.3 of Δ. Non-blockingness holds, as any trace leading to states without actual outputs, is extended with artificial output δ. Anomaly-freedom holds since quiescence denotes the absence of outputs, which cannot be followed by an output. Quiescence reducibility and stability follow from δ-transitions being self-loops: if δ appears in a suspension trace, it may be removed or replicated by taking the self-loop less or more often.

Suspension languages form a bounded semi-lattice: they are partially ordered by inclusion, and no least element exists (as $\emptyset \notin \mathcal{SL}$), but there is a greatest element. We denote this element by L_χ.

Definition 5.3. $L_\chi \overset{\text{def}}{=} \{\sigma \in (A^{\delta})^* \mid \sigma \text{ is not anomalous}\}$

LEMMA 5.4. *L_χ is the greatest suspension language.*

The traces of L_χ are the traces of a chaotic state with explicit quiescence, shown in Figure 3. Such a state is used in [2] as the state that contains all possible suspension traces.

5.1 Implementations as Suspension Languages

In this section, we will show the correspondence between implementations and the class of suspension languages. We already established that the suspension traces of any IOTS are a suspension language.

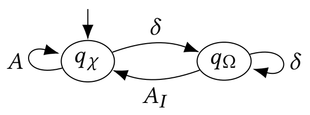

Figure 3: A chaotic state with traces $(q_\chi) = L_\chi$**. Arrows with A and A_I represent sets of transitions for those actions.**

LEMMA 5.5. *For $i \in \mathcal{IOTS}$, we have* Straces $(i) \in \mathcal{SL}$.

Conversely, from a suspension language L, we can construct a canonical IOTS which has exactly the traces of L. We prove this by lifting the canonical specification of Willemse [20] to the level of suspension languages. This canonical specification is based on the Myhill-Nerode equivalence, which is a right-congruence [13].

Definition 5.6. For $L \subseteq (A^{\delta})^*$, the Myhill-Nerode equivalence $\equiv_L \subseteq L \times L$ is defined as

$$\sigma \equiv_L \sigma' \overset{\text{def}}{\Leftrightarrow} \forall \rho \in (A^{\delta})^* : \sigma\rho \in L \iff \sigma'\rho \in L\,.$$

We write $[\sigma]_L$ to denote the equivalence class of $\sigma \in L$.

THEOREM 5.7. *(Myhill-Nerode congruence [13]) Let $L \subseteq (A^{\delta})^*$, and $\sigma, \sigma' \in L$ with $\sigma \equiv_L \sigma'$ and $\sigma\ell \in L$. Then $\sigma'\ell \in L$ and $\sigma\ell \equiv_L \sigma'\ell$.*

The general approach for defining canonical automata for languages is to take every equivalence class $[\sigma]_L$ to be a state, with transitions $[\sigma]_L \xrightarrow{\ell} [\sigma\ell]_L$ for every extension $\sigma\ell$. This gives a minimal, deterministic automaton. For IOTSes, this approach fails, as the suspension language contains occurrences of δ, which cannot be encoded as explicit transitions. Transitions for τ can be used instead. This results in a non-deterministic automaton, so minimality results for canonical deterministic finite automata cannot be lifted in a trivial manner.

Definition 5.8. Let $L \in \mathcal{SL}$. We define the *canonical IOTS* of L to be can(L) with

$$\begin{aligned}
Q_{\text{can}(L)} &\overset{\text{def}}{=} \{[\sigma]_L \mid \sigma \in L\} \\
T_{\text{can}(L)} &\overset{\text{def}}{=} \{([\sigma]_L, \ell, [\sigma\ell]_L) \mid \ell \in A, \sigma\ell \in L\} \\
&\quad \cup \{([\sigma]_L, \tau, [\sigma\delta]_L) \mid \sigma \in L, \sigma \not\equiv_L \sigma\delta\} \\
q^0_{\text{can}(L)} &\overset{\text{def}}{=} [\epsilon]_L
\end{aligned}$$

By Theorem 5.7, the transition relation does not depend on the choice of representatives for equivalence classes and is thus well defined. Furthermore, quiescence reducibility and stability imply that any trace $\sigma\delta \in L$ must have $\sigma\delta \equiv_L \sigma\delta\delta$. This prevents successive τ-transitions, and in particular infinite sequences of τ-transitions. The suspension traces of this IOTS represent exactly the given suspension language.

LEMMA 5.9. *Let $L \in \mathcal{SL}$. Then* Straces (can(L)) $= L$.

Together with Lemma 5.5, we can now define the central result of this section: we can reason about IOTSes as suspension languages, and vice versa. This lifts the Nerode theorem for DFAs and regular languages [13] to IOTSes and suspension languages.

THEOREM 5.10. *Let $L \in (A^{\delta})^*$. Then*

$$L \in \mathcal{SL} \iff \exists i \in \mathcal{IOTS} : \text{Straces}\ (i) = L$$

5.2 Specifications as Suspension Languages

Whereas the correspondence between suspension languages and implementations is clear, the correspondence with specifications is less so. The suspension traces of a specification LTS satisfy all conditions for suspension languages, by the same reasoning as for implementations, except for input-enabledness. For example, specification s_A in Figure 1 contains trace δ but not $\delta a?$.

Lemma 5.11. *[20] For $s \in \mathcal{LTS}$,* Straces (s) *is prefix-closed, non-blocking, anomaly free, quiescence stable and quiescence reducible.*

Next to angelic completion, specifications are often made input-enabled by *demonic completion* [4], i.e., by adding missing input transitions to a chaotic state such as q_χ in Figure 3. If any suspension trace non-deterministically leads to multiple specification states, of which one has a transition for input $a?$ and another does not, then this form of input completion considers that input to be underspecified. For example, trace $a?$ is then underspecified in s_A, and trace $a?\delta$ is then allowed by s_A. This does not correspond to the notion of input underspecification according to ioco, as explained in Example 3.1. We therefore take a different approach, in which we conclude underspecification based on the suspension traces.

Definition 5.12. Let $L \subseteq (A^\delta)^*$ be a language. Then the *input-enabling* of L, denoted by inp(L), is defined as

$$\text{inp}(L) \overset{\text{def}}{=} L \cup \{\sigma a \rho \mid \sigma \in L, a \in A_I, \sigma a \notin L, \rho \text{ is not anomalous}\}$$

Definition 5.13. Let $s \in \mathcal{LTS}$. Then the *Itraces* of s are defined as

$$\text{Itraces}\ (s) \overset{\text{def}}{=} \text{inp}(\text{Straces}\ (s))$$

Remark that the Itraces and Straces of an implementation are the same, as input-enabling does not affect the Straces of an IOTS.

Lemma 5.14. *Let $i \in \mathcal{IOTS}$, $s \in \mathcal{LTS}$. Then*

$$i\ \textbf{ioco}\ s \iff \text{Straces}\ (i) \subseteq \text{Itraces}\ (s)$$

This lemma states that a conforming implementation may only contain the Itraces of a specification, so the Itraces seem to be precisely the conformal traces. This, however, does not take into account the dependencies between traces. We repeat the examples of Section 3, now using the properties of suspension languages of Definition 5.2.

Example 5.15. Trace $\delta a?\delta$ is added to the suspension traces of s_A by transformation inp: the suspension traces contain δ but not $\delta a?$, and extension δ is non-anomalous. Trace $\delta a?\delta$ is therefore in Itraces (s_A), but $a?\delta$ is not. This proves that Itraces (s_A) is not quiescence reducible. We observed in Examples 3.2 and 4.3 that $\delta a?\delta$ is not an ioco-conformal trace of s_A, as implementations cannot contain $\delta a?\delta$ without also containing the non-conformal trace $a?\delta$. This is because implementations are quiescence reducible. More generally, the traces in Itraces (s_A) violating quiescence reducibility are the traces $\delta^+ a?\delta\rho$, for non-anomalous ρ, which are all non-conformal.

Example 5.16. We cast Examples 3.3 and 4.4 to the domain of suspension languages. We prove that s_D is unimplementable, by assuming an implementation i with i **ioco** s_D. This must lead to a contradiction. The suspension traces of i are a suspension language (Theorem 5.2), contained in Itraces (s_D) (Lemma 5.14). Thus, Straces (i) cannot contain the following traces: $a?\delta a?\delta$, $a?\delta a?x!$ and $\delta a?a?y!$ (not in Itraces (s_D)); $\delta a?\delta a?\delta$, $\delta a?\delta a?x!$ and $\delta a?\delta a?y!$ (quiescence reducibility); $\delta a?\delta a?$ (non-blockingness); $\delta a?\delta$ (input-enabledness); $\delta a?x!$ and $\delta a?y!$ (not in Itraces (s_D)); $\delta a?$ (non-blockingness); δ (input-enabledness); $x!$ and $y!$ (not in Itraces (s_D)); and finally, ϵ (non-blockingness).

From $\epsilon \notin$ Straces (i), it follows by prefix-closedness that no trace can be in Straces (i), so it is empty. This cannot occur for any IOTS.

To summarize, the suspension traces of a specification are not a suspension language, because they are not input-enabled. They become input-enabled by transformation inp, but we lose quiescence reducibility.

Lemma 5.17. *Transformation* inp *adds input-enabledness, preserves prefix-closedness, non-blockingness, anomaly freedom, and quiescence stability, but does not preserve quiescence reducibility.*

Although the Itraces of a specification are not a suspension language, they may have a greatest suspension language contained in them, which we may use instead. The suspension traces of all implementations conforming to a specification s are contained in Itraces (s), so their union also is contained in it. Since suspension languages are closed under union, this union is the greatest suspension language contained in s. This suspension language contains precisely all traces of conforming implementations, and is therefore the ioco characterization of s. This always holds, except for unimplementable specifications, as their ioco characterization is empty. As such, the empty language is the only ioco characterization which is not a suspension language.

Definition 5.18. We denote the domain of *ioco characterizations* by $\mathcal{SL}_\emptyset \overset{\text{def}}{=} \mathcal{SL} \cup \{\emptyset\}$.

For $L \subseteq (A^\delta)^*$, we define ic(L) as the greatest ioco characterization included in L.

Lemma 5.19. *Let $s \in \mathcal{LTS}$. Then $\langle s \rangle_{\textbf{ioco}} = \text{ic}(\text{Itraces}\ (s))$*

The domain of ioco characterizations extends the semi-lattice of suspension language to a complete lattice, by adding the least element $\emptyset$. We conclude that every implementable specification has a suspension language as ioco characterization, and thus also has a canonical implementation, through Theorem 5.10. Unimplementable specifications do not. Remark that $\emptyset$ meets all the conditions for being a suspension language, except for being non-empty: any ioco characterization therefore meets these conditions as well.

Figure 4 gives an overview of the introduced transformations for ioco, the relevant properties that (do not) hold after every transformation, and the conformance and refinement relations.

6 LTS CHARACTERIZATIONS OF IOCO

So far, we have found a semantical characterization of specifications in terms of conformal traces, and in terms of constraints on suspension languages. These characterizations have been proven to answer ($Q1$) and ($Q2$), based on canonical IOTSes, and the construction of these canonical IOTSes has been treated. However, to construct the canonical IOTS of Definition 5.8, we require an explicit, syntactical representation of the (equivalence classes of) traces in ic(Itraces (s)). As such, we lift the required transformations to concrete LTS-representations of these trace sets. For finite

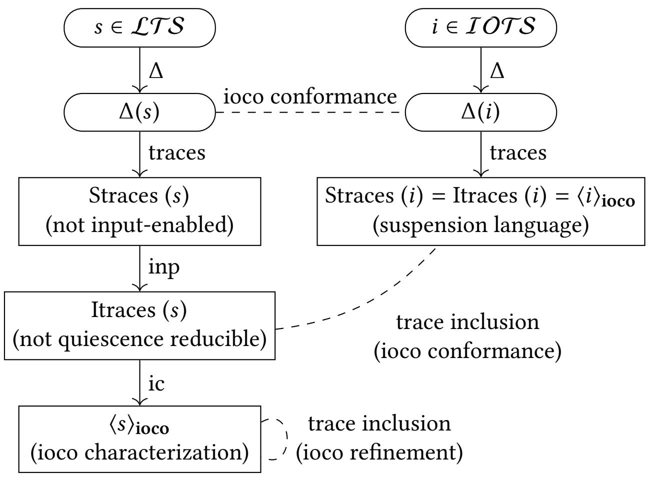

Figure 4: An overview of the introduced trace transformations. Rounded nodes are LTSes, and boxes represent languages with the given properties. Dashed lines represent relations between domains.

$$\begin{tikzcd}
\mathcal{LTS} \arrow[r, "\Delta"] \arrow[d, "\Gamma"'] \arrow[dr, "\text{Straces}"'] & \mathcal{LTS}(A_I, A_U^\delta) \arrow[d, "\text{traces}"] \\
\mathcal{LTS}^\delta \arrow[r, "\text{traces}"'] \arrow[d, "\Xi"'] & (A^\delta)^* \arrow[d, "\text{inp}"] \\
\mathcal{LTS}^\delta \arrow[r, "\text{traces}"'] \arrow[d, "\zeta"'] & (A^\delta)^* \arrow[dd, "\text{ic}"] \\
\mathcal{LTS}^\delta \arrow[d, "\eta"'] & \\
\mathcal{LTS}^\delta \arrow[r, "\text{traces}"] & \mathcal{SL}_\emptyset
\end{tikzcd}$$

Figure 5: Transformations from specification LTSes to ioco characterizations, via LTS-representation (left) and on their traces (right).

specifications, this directly provides algorithmic means for comparing ioco characterizations and for generating canonical IOTSes. For infinite specifications, this gives insight into the computability of ioco characterizations.

A straightforward representation of traces with explicit δ actions, is by a deterministic LTSes with explicit δ-transitions. Note that a deterministic LTS has no internal transitions.

Definition 6.1. $\mathcal{LTS}^\delta \overset{\text{def}}{=} \{s \in \mathcal{LTS}(A_I, A_U^\delta) \mid$ s is deterministic and traces (s) is anomaly free$\}$

Figure 5 gives an overview of the transformations on traces and the corresponding transformations on LTSes: a computation of the Itraces is introduced in Section 6.1, and operation ic is discussed in Section 6.2. We perform an analysis of the properties for suspension languages, which should be obtained or preserved by these transformations. Formally, we want this diagram to commute.

6.1 Itraces of LTSes

We obtain a deterministic LTS representing the suspension traces of specification s by determinizing $\Delta(s)$, using the standard subset construction. This is also known as the *suspension automaton* of s [17]. Its traces are the suspension traces of s.

Definition 6.2. Let $s \in \mathcal{LTS}$. Then the *suspension automaton* $\Gamma(s) \in \mathcal{LTS}^\delta$ is defined by

$$Q_{\Gamma(s)} \overset{\text{def}}{=} \mathcal{P}(Q_{\Delta(s)}) \setminus \emptyset$$
$$T_{\Gamma(s)} \overset{\text{def}}{=} \{(q, \ell, q \text{ after}_{\Delta(s)} \ell) \mid q \in Q_{\Gamma(s)}, \ell \in L^\delta, (q \text{ after}_{\Delta(s)} \ell) \neq \emptyset\}$$
$$q^0_{\Gamma(s)} \overset{\text{def}}{=} q^0_{\Delta(s)} \text{ after } \epsilon$$

Lemma 6.3. *[17] Let $s \in \mathcal{LTS}$. Then* traces $(\Gamma(s))$ = Straces (s).

After performing determinization, the input completion of $\Gamma(s)$ can be obtained by demonic completion [2]. This adds missing input transitions to a chaotic state with explicit quiescence, such as q_χ in Figure 3.

Definition 6.4. Let $s \in \mathcal{LTS}^\delta$, and choose $\chi \in \mathcal{LTS}^\delta$ with traces $(\chi) = L_\chi$ and $Q_s \cap Q_\chi = \emptyset$. Then $\Xi(s) \in \mathcal{LTS}^\delta$ is defined by

$$Q_{\Xi(s)} \overset{\text{def}}{=} Q_s \cup Q_\chi$$
$$T_{\Xi(s)} \overset{\text{def}}{=} T_s \cup T_\chi \cup \{(q, a, q^0_\chi) \mid q \in Q_s, a \in A_I, q \overset{a}{\not\rightarrow})\}$$
$$q^0_{\Xi(s)} \overset{\text{def}}{=} q^0_s$$

The left LTS in Figure 6 shows the result of applying Γ and Ξ on specification s_A of Figure 1. Demonic completion on specifications in $\mathcal{LTS}$ has been studied in the context of ioco [4], but is usually performed without determinization. The approach of Beneš et al. is similar to ours [2], performing demonic completion on determinized suspension automata, but they do not remark that the resulting suspension automaton may violate quiescence reducibility. Instead, they assume input-enabled, quiescence reducible suspension automata, restricting their results. For example, this excludes specification $\Xi(\Gamma(s_A))$.

By Lemmas 6.3 and 6.5, the Itraces of a specification are the traces of $\Xi(\Gamma(s))$. We can consequently use trace inclusion to $\Xi(\Gamma(s))$ to check for ioco conformance to s (Lemma 5.14). For finite specifications (that is, having a finite set of states and transitions), the size of $\Gamma(s)$ is exponential in the size of s [17], and trace inclusion for deterministic specifications is polynomial. This is in line with the known, exponential complexity bounds [14].

Lemma 6.5. *Let $s \in \mathcal{LTS}^\delta$. Then*

(1) traces $(\Xi(s))$ = Itraces (s),

(2) $\Xi(s) \in \mathcal{LTS}^\delta$ holds, that is, $\Xi(s)$ is deterministic and its traces are anomaly free.

6.2 Transformation ic on LTSes

The last step of computing ioco characterizations is by translating transformation ic to LTSes. This is done in two steps, and resembles the construction of *validization* by Beneš et al. [2], as well as transformation D by Bourdonov and Kossatchev [5]. The first step (ζ) ensures quiescence reducibility and stability, while the second (η) ensures non-blockingness.

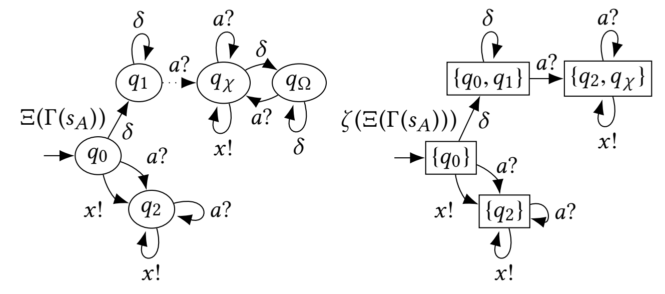

Figure 6: Successive application of Γ and Ξ (left) and ζ (right) on s_A. The dotted transition is added by Ξ.

Definition 6.6. Let $s \in \mathcal{LTS}^\delta$. Then we define $\zeta(s) \in \mathcal{LTS}^\delta$ by

$$Q_{\zeta(s)} \stackrel{\text{def}}{=} \mathcal{P}(Q_s) \setminus \{\emptyset\}$$

$$T_{\zeta(s)} \stackrel{\text{def}}{=} \{(r, \ell, r \text{ after}_s\, \ell) \mid r \in Q_{\zeta(s)},\ \ell \in A,\ \forall q \in r : q \overset{\ell}{\Longrightarrow}_s\}$$

$$\cup \{(r, \delta, \Re(r)) \mid r \in Q_{\zeta(s)},\ \forall q \in r, \forall n \in \mathbb{N} : q \overset{\delta^n}{\Longrightarrow}_s\}$$

$$\text{where } \Re(r) \stackrel{\text{def}}{=} \bigcup\{r \text{ after}_s\, \delta^n \mid n \in \mathbb{N}\}$$

$$q^0_{\zeta(s)} \stackrel{\text{def}}{=} \{q^0_s\}$$

In the construction of $\zeta(s)$, a singleton state $\{q\}$ has the same behaviour as q in the original LTS s for actions in A. The behaviour for δ is changed: δ-transitions in s may cause $\zeta(s)$ to move from a singleton to a non-singleton state. Intuitively, $\zeta(s)$ after σ is the set of states of s which are reached by σ, or by any trace created by removing or duplicating δ-occurrences from σ. A trace is not allowed by such a set in $\zeta(s)$, if any state of this set does not allow it in s. This ensures quiescence reducibility.

Example 6.7. The right LTS in Figure 6 results from applying ζ to $\Xi(\Gamma(s_A))$. From the initial state, after a or x, the behaviour is not changed with respect to $\Xi(\Gamma(s_A))$ itself. We find $\Re(\{q_0\}) = \{q_0, q_1\}$, so $\{q_0\} \xrightarrow{\delta}_{\zeta(\Xi(\Gamma(s_A)))} \{q_0, q_1\}$. This is because trace δ reaches q_1, but trace ϵ (which is trace δ with a δ-occurrence omitted) reaches q_0. State $\{q_0, q_1\}$ is similar to q_1 in $\Xi(\Gamma(s_A))$, except that the latter contains trace $a\delta$, whereas the former does not, since $q_0 \overset{a\delta}{\not\Longrightarrow}$. Consequently, unimplementable trace $\delta a?\delta$ is removed by ζ. This yields a quiescence reducible LTS, trace equivalent to specification $\Delta(s_C)$.

Example 6.8. Figure 7 shows ζ applied on $\Xi(\Gamma(s_D))$, for s_D in Figure 2. States of $\Gamma(s_D)$ have been renamed for readability. In s_D itself, $\delta a?\delta a?$ is underspecified, so this trace leads to q_χ in $\Xi(\Gamma(s_D))$. If δ-occurrences are removed from this trace, states 3, 8 and 9 can be reached, so $\zeta(\Xi(\Gamma(s_D)))$ after $\delta a?\delta a? = \{3, 8, 9, q_\chi\}$. This state allows only outputs allowed by each of the individual states. All outputs are allowed by q_χ, but states 8 and 9 allow respectively $\{y!\}$ and $\{x!\}$. The intersection is empty, so $\{3, 8, 9, q_\chi\}$ is blocking.

Lemma 6.9. *Let $s \in \mathcal{LTS}^\delta$. Then $\zeta(s) \in \mathcal{LTS}^\delta$, and the largest quiescence reducible and stable subset of* traces (s) *is* traces $(\zeta(s))$.

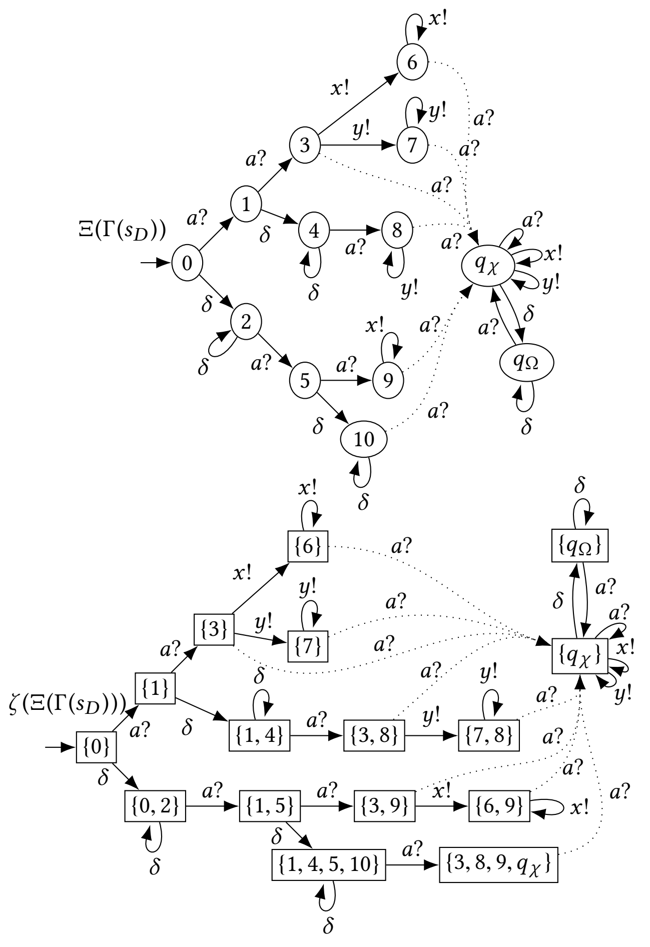

Figure 7: Successive application of Γ and Ξ (above) and ζ (below) on specification s_D, for $A_I = \{a?\}$ and $A_U = \{x!, y!\}$. Transitions to q_χ and $\{q_\chi\}$ are dotted to improve readability.

Transformation ζ is used to regain quiescence reducibility, which may be lost after performing Ξ. In turn, ζ does not preserve non-blockingness, shown by state $\{3, 8, 9, q_\chi\}$ in Example 6.8. We use the *pruning*-procedure introduced in [2] to solve this.

Definition 6.10. Let $s \in \mathcal{LTS}^\delta$. Then the set of *invalid states*, denoted by $\text{inv}(s) \subseteq Q_s$, is defined as the smallest set of states $q \in Q_s$, for which

- $\exists a \in A_I : q \text{ after } a \subseteq \text{inv}(s)$, or
- $\forall x \in \text{out}(q) : q \text{ after } x \subseteq \text{inv}(s)$.

If $q^0_s \in \text{inv}(s)$, then we define $\eta(s) \stackrel{\text{def}}{=} \perp$ with traces $(\perp) \stackrel{\text{def}}{=} \emptyset$.
If $q^0_s \notin \text{inv}(s)$, then we define $\eta(s) \in \mathcal{LTS}^\delta$ by

$$Q_{\eta(s)} \stackrel{\text{def}}{=} Q_s \setminus \text{inv}(s)$$

$$T_{\eta(s)} \stackrel{\text{def}}{=} T_s \cap (Q_{\eta(s)} \times A^\delta \times Q_{\eta(s)})$$

$$q^0_{\eta(s)} \stackrel{\text{def}}{=} q^0_s$$

Note that a blocking state serves as an inductive basis for inv, as it vacuously satisfies the second condition for being invalid.

Intuitively, η removes all blocking states and transitions to and from those states. If this causes any new states to become blocking or non-input-enabled, we recursively remove these as well. The traces of the resulting LTS (or $\bot$) are non-blocking, and preserve quiescence stability and reducibility.

Lemma 6.11. *Let $s \in \mathcal{LTS}^\delta$. Then*

(1) if $q_s^0 \notin \text{inv}(s)$, then indeed $\eta(s) \in \mathcal{LTS}^\delta$,
(2) traces $(\eta(s))$ is the largest input-enabled and non-blocking subset of traces (s), and
(3) traces $(\eta(s))$ is quiescence stable or reducible, if traces (s) is, respectively.

Theorem 6.12. *Let $s \in \mathcal{LTS}$. Then*

$$\langle s \rangle_{\textbf{ioco}} = \text{traces}\ (\eta(\zeta(\Xi(\Gamma(s)))))$$

Example 6.13. In Figure 6, $\zeta(\Xi(\Gamma(s_A)))$ only has non-blocking states. Consequently, η leaves it unchanged, which means that $\langle s_A \rangle_{\textbf{ioco}} = \text{traces}\ (\zeta(\Xi(\Gamma(s_A))))$. Since s_C is an IOTS, we have $\langle s_C \rangle_{\textbf{ioco}} = \text{Straces}\ (s_C) = \text{traces}\ (\Delta(s_C))$. As $\zeta(\Xi(\Gamma(s_A)))$ is trace equivalent to $\Delta(s_C)$, we thus have $\langle s_A \rangle_{\textbf{ioco}} = \langle s_C \rangle_{\textbf{ioco}}$. By Theorem 4.5.4, specifications s_A and s_C are thus proven to be equivalent.

Example 6.14. To find the ioco characterization of s_D, we apply η on $\zeta(\Xi(\Gamma(s_D)))$, in Figure 7. Blocking state $\{3, 8, 9, q_\chi\}$ is invalid, as well as states $\{1, 4, 5, 10\}$, $\{1, 5\}$, $\{0, 2\}$ and initial state $\{0\}$. Hence, the result is $\bot$, so $\langle s_D \rangle_{\textbf{ioco}} = \emptyset$.

All trace operations in Figure 4 are now instantiated by transformations on LTSes. Canonical IOTSes can be created from this using Definition 5.8. Ioco refinement and equivalence can be checked by trace inclusion on transformed specifications, using Theorems 4.5.3, 4.5.4 and 6.12.

6.3 Complexity and Undecidability

As both Γ and ζ are exponential, the construction of the LTS-representation has a double exponential upper bound. Since operation ζ is required to remove nonconformal traces from the Itraces, we conjecture that deciding refinement and equivalence for finite specifications is not in PSPACE. The relevance of this complexity is limited, because specifications are often (practically) infinite. For example, they can be represented by process algebras [18] or symbolic transition systems [6], containing data parameters. Computing explicit LTS representations is therefore often infeasible.

A more feasible approach for infinite systems is to check whether individual traces are conformal. This does not allow checking equivalence or refinement, but it would allow comparing finite parts of specifications. Furthermore, a conforming implementation can then be derived in a lazy manner. For example, one could build a 'simulator' for a specification which behaves like a conforming implementation, by producing conformal traces.

Unfortunately, deciding whether traces are conformal is undecidable, even if the after-set q after ℓ of every state q and action ℓ is finite and computable.[1] Any trace σ may be nonconformal because some extension $\sigma\rho$ of unknown length is blocking. In general, the entire state space of the specification must be checked for blocking states, by construction of η, to detect nonconformal traces.

[1]Without this assumption, incomputability is trivial, as a single step q after ℓ may already be incomputable. [3] describes a formal definition of computability of after-sets.

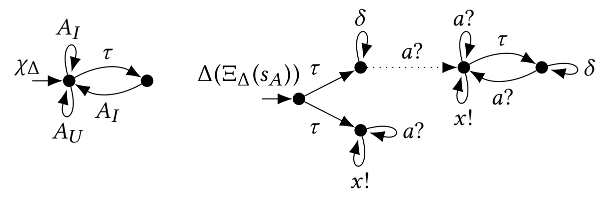

Figure 8: A chaotic state without explicit quiescence, and demonic completion Ξ_Δ of s_A.

Theorem 6.15. *Let $\sigma \in (A^\delta)^*$, and let $s \in \mathcal{LTS}$ where T_s is finitely branching, and Q_s, A_I and A_U are enumerable, and where $\text{init}(q)$ and q $\text{after}_{\Delta(s)}\ \ell$ are computable for all $\ell \in A^\delta$ and all $q \in Q_s$. Determining whether σ* **iocfl** *s holds is undecidable.*

7 UIOCO REFINEMENT

The need for constructions ζ and η arises from the problems with input underspecification, in combination with non-determinism. We now repeat the analysis for the alternative conformance relation uioco [4]. Whereas in ioco, allowed suspension traces are made explicit by performing demonic completion *after* adding quiescence and determinization (transformation Γ), uioco performs demonic completion on the initial specification, *before* applying Γ. Therefore, a chaotic state without explicit quiescence and with internal transitions is used [4], such as the initial state of χ_Δ in Figure 8. This explicitly transforms an LTS specification to an IOTS.

Definition 7.1. Let $s \in \mathcal{LTS}$, and choose $\chi_\Delta \in \mathcal{IOTS}$ with $\text{Straces}\ (\chi_\Delta) = L_\chi$ and $Q_s \cap Q_{\chi_\Delta} = \emptyset$. Then $\Xi_\Delta(s) \in \mathcal{IOTS}$ is defined by

$$Q_{\Xi_\Delta(s)} \overset{\text{def}}{=} Q_s \cup Q_{\chi_\Delta}$$
$$T_{\Xi_\Delta(s)} \overset{\text{def}}{=} T_s \cup T_{\chi_\Delta} \cup \{(q, a, q^0_{\chi_\Delta}) \mid q \in Q_s, a \in A_I, q \overset{a}{\not\Rightarrow}\}$$
$$q^0_{\Xi_\Delta(s)} \overset{\text{def}}{=} q^0_s$$

Definition 7.2. Let $i \in \mathcal{IOTS}$ and $s \in \mathcal{LTS}$. Then

$$i\ \textbf{uioco}\ s \quad \overset{\text{def}}{\Leftrightarrow} \quad i\ \textbf{ioco}\ \Xi_\Delta(s)$$

The right LTS in Figure 8 results from demonic completion Ξ_Δ. It shows that for uioco, s_A allows suspension trace $a?\delta$, whereas this is not the case for ioco. In general, uioco is weaker than ioco [4]. We define characterizations similarly to for ioco.

Definition 7.3. Let $\sigma \in (A^\delta)^*$ and $s, s' \in \mathcal{LTS}$. Then

$$\sigma\ \textbf{uiocfl}\ s \overset{\text{def}}{\Leftrightarrow} \exists i \in \mathcal{IOTS} : i\ \textbf{uioco}\ s \wedge \sigma \in \text{Straces}\ (i)$$
$$\langle s \rangle_{\textbf{uioco}} \overset{\text{def}}{=} \{\sigma \in (A^\delta)^* \mid \sigma\ \textbf{uiocfl}\ s\}$$
$$s \preccurlyeq_u s' \overset{\text{def}}{\Leftrightarrow} \forall i \in \mathcal{IOTS} : i\ \textbf{uioco}\ s \implies i\ \textbf{uioco}\ s'$$

Since every specification s is explicitly transformed to an IOTS, uioco-conformance to s is equivalent to suspension trace inclusion to $\Xi_\Delta(s)$, by Lemma 5.14. This entails that the suspension traces of the completed specification act directly as the uioco characterization. Consequently, the domain of uioco-characterizations is $\mathcal{SL}$. Moreover, $\Xi_\Delta(s)$ acts directly as a canonical implementation of specification s, as it is uioco-equivalent to s.

Theorem 7.4. *Let $s, s' \in \mathcal{LTS}$ and $i \in \mathcal{IOTS}$. Then*

(1) $\langle i \rangle_{\mathbf{uioco}} = \text{Straces}\,(i)$
(2) $\langle s \rangle_{\mathbf{uioco}} = \text{Straces}\,(\Xi_\Delta(s))$
(3) $\langle s \rangle_{\mathbf{uioco}} \in \mathcal{SL}$
(4) $i\ \mathbf{uioco}\ s \iff \langle i \rangle_{\mathbf{uioco}} \subseteq \langle s \rangle_{\mathbf{uioco}}$
(5) $s \preccurlyeq_u s' \iff \langle s \rangle_{\mathbf{uioco}} \subseteq \langle s' \rangle_{\mathbf{uioco}}$

Hence, we can check uioco conformance or refinement directly by checking suspension trace inclusion, after performing demonic completion on the specification. This is PSPACE-complete [15].

In contrast to the ioco-characterization, this characterization is decidable. Intuitively, to check whether a suspension trace σ is conformal to s, we traverse s by following σ. We do this inductively, per action. If an output contained in σ leads to an empty set of states, then it is forbidden and σ is not allowed. If any input contained in σ is underspecified, then σ is allowed. If we encounter no such forbidden outputs or underspecification inputs, then σ is allowed.

Theorem 7.5. *Let $\sigma \in (A^\delta)^*$, and let $s \in \mathcal{LTS}$ where T_s is finitely branching, and Q_s, A_I and A_U are enumerable, and where $\text{init}(q)$ and $q\ \text{after}_{\Delta(s)}\ \ell$ are computable for all $\ell \in A^\delta$ and all $q \in Q_s$. Determining whether σ **uiocfl** s holds is decidable.*

8 DISCUSSION AND CONCLUSIONS

We answered ($Q1$): given a specification, we can construct a canonical conforming implementation using the ioco-characterization set of the specification, if this set is not empty. If it is empty, no conforming implementation exists. We also answered ($Q2$): we defined an ioco-refinement preorder for specification models, such that a refined model allows less conforming implementations. Both answers were first given declaratively, i.e., defined in terms of properties, and were then turned into a constructive form.

We established a computable refinement preorder for both ioco and uioco. The characterization for ioco forms a complete lattice, as the empty characterization can be expressed by actual specifications. This paves the way for using results from lattice theory in the context of ioco theory. For example, the *merge* operator, introduced by Beneš et al. [2], acts as the greatest lower bound in this lattice, which we can now express as language intersection. In that work, an artificial unimplementable specification is introduced, but this is unneeded: example specification s_D acts as one. For uioco, no unimplementable specification exists, so uioco specifications form a semi-lattice without a least element. Defining a merge operator for uioco thus requires adding such a least element artificially.

The combination of quiescence and trace-based input underspecification in ioco has unexpected consequences for the ioco characterization, such as the high computational complexity for finite specifications, and undecidability in general. In particular, simulating an ioco-conforming implementation is not possible. This treatment of underspecified inputs and quiescence is found in many variants of ioco, such as modal ioco [12], symbolic ioco [6] and probabilistic ioco [8], so we expect similar consequences for these relations. To overcome these problems, a uioco-like approach may be possible for these variants as well. Demonic completion should then be lifted to the level of modal, symbolic or probabilistic transition systems. If quiescence is abandoned in favour of an explicit notion of time, such as with tioco [11], implications are less clear.

In this paper, we did not touch upon test case generation, but this is a core motivation of model based testing. After observing a suspension trace, the verdict for a test case should ideally be *pass* for conformal traces, and *fail* for nonconformal traces. Theorem 6.15 shows that this ideal test case generation is impossible for ioco. Standard test case generation for ioco [18] is weaker, as non-conformal traces do not always fail [5]. For example, trace $\delta a?\delta$ does not fail when testing for specification s_A, and neither does trace ϵ when testing for s_D. In contrast, Theorem 7.5 shows that such an ideal test case generation is possible for uioco.

This adds up to the favourable compositionality properties of uioco, for which it was originally introduced [4]. One could thus argue that uioco has a more sensible semantics than ioco, and therefore should be used as the standard conformance relation in testing theory for LTSes with inputs, outputs and quiescence.

REFERENCES

[1] L. de Alfaro and T.A. Henzinger. Interface Automata. In V. Gruhn, editor, *ESEC/FSE'01*, SIGSOFT Softw. Eng. Notes 26, pages 109–120. ACM Press, 2001. URL http://doi.acm.org/10.1145/503271.503226.

[2] N. Beneš, P. Daca, T.A. Henzinger, J. Křetínský, and D. Ničković. Complete composition operators for ioco-testing theory. In P. Kruchten et al., editors, *ACM SIGSOFT Symp. on Comp.-Based Softw. Eng.*, pages 101–110. ACM, 2015. doi: 10.1145/2737166.2737175. URL http://doi.acm.org/10.1145/2737166.2737175.

[3] JA Bergstra, JCM Baeten, and Jan Willem Klop. On the consistency of Koomen's fair abstraction rule. *TCS*, 37(1):129–176, 1987.

[4] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional Testing with IOCO. In A. Petrenko and A. Ulrich, editors, *FATES 2003*, LNCS 2931, pages 86–100. Springer-Verlag, 2004. doi: http://dx.doi.org/10.1007/978-3-540-24617-6_7.

[5] I.B. Bourdonov and A.S. Kossatchev. Specification completion for IOCO. *Programming and Computer Softw.*, 37(1):1–14, 2011. doi: 10.1134/S0361768811010014. URL https://doi.org/10.1134/S0361768811010014.

[6] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund et al., editors, *FATES/RV'06*, LNCS 4262, pages 40–54. Springer-Verlag, 2006. doi: http://dx.doi.org/10.1007/11940197_3.

[7] M.-C. Gaudel. Testing can be Formal, too. In P.D. Mosses et al., editors, *TAPSOFT'95*, LNCS 915, pages 82–96. Springer-Verlag, 1995.

[8] Marcus Gerhold and Mariëlle Stoelinga. Model-based testing of probabilistic systems. *Formal Aspects of Computing*, 30(1):77–106, 2018. doi: 10.1007/s00165-017-0440-4. URL https://doi.org/10.1007/s00165-017-0440-4.

[9] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[10] R. Janssen and J. Tretmans. Matching implementations to specifications: The corner cases of ioco. Technical report, 2019. URL https://sumbat.cs.ru.nl/Publications?action=upload&upname=JanssenTretmansCornerCasesOfIoco.pdf.

[11] M. Krichen and S. Tripakis. Black-Box Conformance Testing for Real-Time Systems. In S. Graf and others., editors, *SPIN'04*, LNCS 2989. Springer-Verlag, 2004.

[12] Malte Lochau, Sven Peldszus, Matthias Kowal, and Ina Schaefer. Model-based testing. In M. Bernardo et al., editors, *Formal Methods for the Design of Computer, Communication and Softw. Systems*, 2014.

[13] Anil Nerode. Linear automaton transformations. *Proc. of the American Mathematical Society*, 9(4):541–544, 1958.

[14] N. Noroozi. *Improving Input-Output Conformance Testing Theories.* PhD thesis, Eindhoven University of Technology, Eindhoven (NL), 2014.

[15] Larry J Stockmeyer and Albert R Meyer. Word problems requiring exponential time. In *Proc. 5th ACM Symp. on Theory of computing*, pages 1–9. ACM, 1973.

[16] G. Stokkink, M. Timmer, and M. Stoelinga. Talking quiescence: a rigorous theory that supports parallel composition, action hiding and determinisation. In A. K. Petrenko and H. Schlingloff, editors, *MBT'12*, volume 80 of *EPTCS*, 2012.

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

[18] J. Tretmans. Model Based Testing with Labelled Transition Systems. In R.M. Hierons et al., editors, *Formal Methods and Testing*, LNCS 4949, pages 1–38. Springer-Verlag, 2008. doi: http://dx.doi.org/10.1007/978-3-540-78917-8_1.

[19] M. Volpato and J. Tretmans. Towards Quality of Model-Based Testing in the IOCO Framework. In *JAMAICA'13*, pages 41–46, New York, NY, USA, 2013. ACM. doi: http://doi.acm.org/10.1145/2489280.2489293.

[20] T.A.C. Willemse. Heuristics for IOCO-Based Test-Based Modelling. In L. Brim et al., editors, *FMICS/PDMC'07*, LNCS 4346, pages 123–147. Springer-Verlag, 2007. URL http://dx.doi.org/10.1007/978-3-540-70952-7_9.