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:

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

We thank Frits Vaandrager for his valuable feedback.

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 (input-output-conformance) 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 quasitransitivity 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?
(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. Bené 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 Bené 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 \((q, \tau, q')\), \(\ldots\) of \(\tau\)-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 \(q_0^s\) 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 to express sets of traces as for formal languages: \(\sigma_1 \cdot \sigma_2\) or \(\sigma_1 \sigma_2\) denotes concatenation; \(\sigma^*\) denotes repetition; \(\sigma^\epsilon\) denotes the Kleene-star, with \(\sigma^* = \sigma^\epsilon\); and \(\epsilon\) 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 \(\sigma_1, \sigma_2 \in (A \cup \{\tau\})^*\). Then we define

\[
\begin{align*}
q \overset{\ell, \sigma}{\longrightarrow} q' & \overset{\text{def}}{=} (\sigma, \ell) \in Q \times (A \cup \{\tau\})^* \\
q \overset{\sigma}{\longrightarrow} q'' & \overset{\text{def}}{=} \exists q' \in Q_s: q \overset{\sigma}{\longrightarrow} q'' \land (q'', q') \in T_s \\
q \overset{q_1 \cdot \sigma}{\longrightarrow} q' & \overset{\text{def}}{=} \exists n \in \mathbb{N}: q \overset{q_1}{\longrightarrow} q^n \overset{\sigma}{\longrightarrow} q' \\
q \overset{q_1 \cdot \sigma}{\longrightarrow} q' & \overset{\text{def}}{=} \exists q_1, q_2 \in Q_s: q \overset{\sigma}{\longrightarrow} q_1 \overset{\epsilon}{\longrightarrow} q_2 \overset{q'}{\longrightarrow} q' \\
q \overset{q_1 \cdot \sigma}{\longrightarrow} q' & \overset{\text{def}}{=} q \overset{q'}{\longrightarrow} q' \\
\text{traces (q)} & \overset{\text{def}}{=} \{\sigma \in A^* | q \overset{\sigma}{\longrightarrow} q\} \\
\text{traces (s)} & \overset{\text{def}}{=} \text{traces (q_0^s)} \\
\text{traces (s) after } & \overset{\text{def}}{=} \text{traces (q_0^s) after } \sigma
\end{align*}
\]
traces \( (Q) \overset{def}{=} \bigcup \{ \text{traces } (q) \mid q \in Q \} \)

\( Q \text{ after } \sigma \overset{def}{=} \bigcup \{ q \text{ after } \sigma \mid q \in Q \} \)

init\((Q) \overset{def}{=} \{ \ell \in A \mid \exists q \in Q : q \xrightarrow{\ell} \} \)

out\((Q) \overset{def}{=} \text{init}(Q) \cap A_U \)

\( s \text{ is input-enabled } \overset{def}{=} \forall q \in Q_s : \forall a \in A_I : q \xrightarrow{a} \)

\( s \text{ is deterministic } \overset{def}{=} \forall q \in Q_s : \forall r \in \text{traces } (q) : q \text{ after } \sigma = 1 \)

\( IOTS \overset{def}{=} \{ s \in LTS \mid s \text{ is input-enabled} \} \)

To avoid ambiguity, we sometimes add the name of an LTS as a subscript to the introduced notation, e.g. we distinguish \( q \text{ after } \sigma, a \) if \( q \in Q_s \) and \( q \in Q_t \). Our domain of implementations is \( IOTS \), 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 \( \delta \not\in A \cup \{ \tau \} \), representing quiescence. We fix the sets of actions \( A_0^\delta = A_U \cup \{ \delta \} \) and \( A^\delta = A \cup \{ \delta \} \). The notation of Definition 2.2 is also used with \( \delta \) as an output.

**Definition 2.3.** Let \( s \in LTS(A_I, A_U) \), and \( q \in Q_s \). Then

\( q \text{ is quiescent in } s \overset{def}{=} \neg \exists \ell \in A_U \cup \{ \tau \} : q \xrightarrow{\ell} \)

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

\[ Q_{\Delta(s)} \overset{def}{=} Q_s \]

\[ T_{\Delta(s)} \overset{def}{=} T_s \cup \{ (q, \delta, q) \mid q \in Q_s, q \text{ is quiescent in } s \} \]

\[ q_{\Delta(s)}^0 \overset{def}{=} q_s^0 \]

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

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 \( i \in IOTS \), \( s \in LTS \). Then

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

Since \( IOTS \subseteq LTS \), 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 \( i, i' \in IOTS \), \( s \in LTS \). Then

\( i \text{ ioco } i' \overset{\Leftrightarrow}{=} \text{Straces } (i) \subseteq \text{Straces } (i') \)

\( i \text{ ioco } i' \land i' \text{ ioco } s \implies i \text{ ioco } s \) (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 \( s, s' \in LTS \). Then

\( s \ll s' \overset{def}{=} \forall i \in IOTS : i \text{ ioco } s \implies i \text{ ioco } s' \)

\( s \simeq s' \overset{def}{=} s \ll s' \land s' \ll s \)

### 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 \( \text{out}(\Delta(s_B) \text{ after } a?) = [\delta, x!] \not\subseteq \text{out}(\Delta(s_A) \text{ after } a?) = [x!] \). 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: \( s_C \text{ ioco } s_A \).

**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 implementability 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 \( \delta \)-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 \( i \text{ ioco } s_A \) 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 \( a? \delta \).
which behaviour is forbidden. Behaviour of implementations is

when considering some form of reflexivity.

As a first step towards formalizing the examples in the previous sec-

occur in any conforming implementation, as was illustrated in

Example 3.3 (Implementability). Consider $s_D$ in Figure 2, intro-

duced by Bourdonov and Kossatchev [5]. Assume an implementa-

tion $i \in IOTS$ with $i \text{ioco } s_D$. We have that $\delta a?\delta a? \in \text{Straces } (i)$,

because:

(1) $\epsilon \in \text{Straces } (i)$; (2) then $\text{out}(\Delta (i) \text{after } \epsilon) \neq \emptyset$; moreover,

$\text{out}(\Delta (s_D) \text{after } \epsilon) = \{\delta\}$, so it must be that $\text{out}(\Delta (i) \text{after } \epsilon) = \{\delta\}$,

and consequently $\delta \in \text{Straces } (i)$; (3) then also $\delta a? \in \text{Straces } (i)$

since $i$ is input-enabled; (4) analogous to (2): $\text{out}(\Delta (s_D) \text{after } \delta a?) = \{\delta\}$, thus $\text{out}(\Delta (i) \text{after } \delta a?) = \{\delta\}$, so also $\delta a? \in \text{Straces } (i)$; (5)

generalizes to (3): $\delta a?\delta a? \in \text{Straces } (i)$, as $i$ is input-enabled.

Let $\text{out}(\Delta (i) \text{after } \delta a?\delta a?) = X$, then, because $\delta a?\delta a? \in \text{Straces } (i)$
it holds that $X \neq \emptyset$ (since there is either a 'real' output $x!$ or $y!$, and

if not, there is output $\delta$). Moreover, since $\delta$-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 $\delta a?\delta a?$:

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

Furthermore, if $i \text{ioco } s_D$ holds, then we must have that:

$\text{out}(\Delta (i) \text{after } \delta a?\delta a?) \subseteq \text{out}(\Delta (s_D) \text{after } \delta a?\delta a?) = \{x!\}$, and, likewise,

$\text{out}(\Delta (i) \text{after } a?\delta a?) \subseteq \text{out}(\Delta (s_D) \text{after } a?\delta a?) = \{y!\}$.

Combining all these constraints for $X$, we conclude that there

is no possible $X$ satisfying all of them. This implies that the con-

forming implementation $i$ cannot exist: $s_D$ is a specification which has no

conforming implementations at all. Apparently, there exist unim-

plementable 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 sec-

tion, 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 $s \in LOTS$. Then

$s \text{ioco } \sigma \iff \exists i \in IOTS : i \text{ioco } s \land \sigma \in \text{Straces } (i)$

$s \text{ioco } \sigma \iff \{\sigma \in (A^\delta)^* \mid \sigma \text{ioco } s\}$

We call $\langle s \rangle \text{ioco }$ 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 the trace $a?\delta$, which is a

nonconformal trace of $s_A$. No conforming implementation $i$

contains this trace because, if it would, it would also have: $\delta \in

\text{out}(\Delta (i) \text{after } a?) \not\subset \text{out}(\Delta (s_A) \text{after } a?) = \{x!\}$.

Example 4.3 (Implementation freedom). In Example 3.2 we argued

that the trace $\delta a?\delta a?$ 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 $\delta a?$,

then, because $\delta$-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 $s_A \not\equiv s_C$, can be proved by

showing that $(s_A)\text{ioco } \not\equiv (s_C)\text{ioco }$

Example 4.4 (Implementability). Specification $s_D$ in Example 3.3
does not have any conforming implementations, so it also has no

conformal traces: $(s_D)\text{ioco } = \emptyset$.

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 $i \in IOTS$ and $s', s \in LOTS$.

(1) $\langle s \rangle \text{ioco } = \text{Straces } (i)$

(2) $\langle s \rangle \text{ioco } \subseteq \langle s' \rangle \text{ioco } \iff i \text{ioco } s'$

(3) $\langle s \rangle \text{ioco } \subseteq \langle s' \rangle \text{ioco } \iff s \leq s'$

(4) $\langle s \rangle \text{ioco } = \langle s' \rangle \text{ioco } \iff s \equiv s'$

Question (Q1), construction of a canonical conforming implement-

ation $i_s$ from a specification $s$, can be answered by taking

$i_s \in IOTS$ such that $\text{Straces } (i_s) = \langle s \rangle \text{ioco }$. According to Theo-

rem 4.5.1 and 4.5.2 we then have $i_s \text{ioco } s$. As Examples 3.3 and 4.4

show, this approach is only possible if $s$ is implementable, that

is, if $(s)\text{ioco } \neq \emptyset$. 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 imple-

mentation 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 construc-

tive descriptions of conformal-trace sets and ioco-characterizations.

Theorem 4.5.1 shows that this will be relatively easy for input-

enable implementations. For specifications, however, it is more
intricate and involves both adding to and removing traces from Straces (σ), 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, whether the traces of such an LTS are the suspension traces of any IOTS. 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 LTS. This characterization leads to a correspondence between implementations and languages with quiescence.

Definition 5.1. A trace σ ∈ (Aδ)∗ is anomalous if it contains an output x following δ, that is, if σ = σ₁δσ₂ for some σ₁, σ₂ ∈ (Aδ)∗ and x ∈ Ω₁.

Definition 5.2. In the following definitions, let σ and ρ range over (Aδ)∗ and let ℓ range over Aδ. A language L ⊆ (Aδ)∗ is
- prefix-closed ⇔ ∀ℓ ∈ L : σ ∈ L
- input-enabled ⇔ ∀σ ∈ L : ∃a ∈ A₁ : σa ∈ L,
- non-blocking ⇔ ∀σ ∈ L : ∃x ∈ A₀ : σx ∈ L
- anomaly free ⇔ ∀σ ∈ L : σ is not anomalous
- quiescence reducible ⇔ ∀σδρ ∈ L : σ ∈ L
- quiescence stable ⇔ ∀σδρ ∈ L : σδδρ ∈ L

Language L is a suspension language if L ≠ ∅ and if all of the above holds. The domain of suspension languages is denoted by SL.

Prefix-closedness and non-emptyness hold for the traces of any LTS. Input-enabledness corresponds to the equally named property on LTSes, and holds for any IOTS. The remaining properties arise from Definition 2.3 of A. Non-blockiness 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 ∅ ∈ SL), but there is a greatest element. We denote this element by L₅.

Definition 5.3. L₅ = {σ ∈ (Aδ)∗ | σ is not anomalous}

Lemma 5.4. L₅ is the greatest suspension language.

The traces of L₅ 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 (q₁) = L₅. Arrows with A and A₁ represent sets of transitions for those actions.

Lemma 5.5. For i ∈ IOTS, we have Straces (i) ∈ 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 ⊆ (Aδ)∗, the Myhill-Nerode equivalence ≡L ⊆ L × L is defined as

σ ≡L σ’ ⇔ ∀ρ ∈ (Aδ)∗ : σρ ∈ L ⇐⇒ σ’ρ ∈ L.

We write [σ]L to denote the equivalence class of σ ∈ L.

Theorem 5.7. (Myhill-Nerode congruence [13]) Let L ⊆ (Aδ)∗, and σ, σ’ ∈ L with σ ≡L σ’ and στ ∈ L. Then στ ∈ L and στ ≡L σ’τ.

The general approach for defining canonical automata for languages is to take every equivalence class [σ]L, to be a state, with transitions [σ]L → [στ]L for every extension στ. 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 ∈ SL. We define the canonical IOTS of L to be can(L) with

Qcan(L) ≡ ([σ]L | σ ∈ L)
Tcan(L) ≡ (∪ L, [σℓ]L | ℓ ∈ A, σL ∈ L)
∪ (∪ [σ]₅, [σδ]₅ | σ ∈ L, σ ≠ L, σδδ ∈ L)
φcan(L) ≡ [ε]L.

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 σδ ∈ L must have σδ ≡L σδδ. 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 ∈ 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 ∈ (Aδ)∗. Then

L ∈ SL ⇐⇒ ∃i ∈ IOTS : 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 $\delta$ but not $\delta a$?

**Lemma 5.11.** [20] For $s \in LOTS$, $Itraces (s)$ 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 $q_x$ 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 $a\delta A$, 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

$$\text{inp}(L) = L \cup \{\sigma a \rho \mid \sigma \in L, a \in A_1, \sigma a \notin L, \rho \text{ is not anomalous}\}$$

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

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

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 $i \in IOTS$, $s \in LTS$. Then

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

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 $\text{inp}$: the suspension traces contain $\delta$ but not $\delta a?$, and extension $\delta$ 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 $\rho$, 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 $i \equiv \text{ioco } s_D$. 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 ($s_D$)); $\delta a?\delta a?\delta$, $\delta a?a?x!$ and $\delta a?\delta a?y!$ (quiescence reducibility); $\delta a?\delta a? \notin \text{(non-blockingness)}$; $\delta a?\delta$ (input-enabledness); $\delta a?x!$ and $\delta a?y!$ (not in Itraces ($s_D$)); $\delta a?\delta$ (non-blockingness); $\delta$ (input-enabledness); $x!$ and $y!$ (not in Itraces ($s_D$)); and finally, $\epsilon$ (non-blockingness).

From $\epsilon \notin \text{Straces } (i)$, 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 $S\subseteq LOTS \cup \{\emptyset\}$. For $L \subseteq (A^\delta)^*$, we define $ic(L)$ as the greatest ioco characterization included in $L$.

**Lemma 5.19.** Let $s \in LTS$. Then $ic(s) = \text{ic(Straces } (s))$

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 ICOO

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
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 $\Gamma(s) \in \mathcal{LTS}^S$ is defined by

$$\Gamma(s) = \langle Q, \delta, q_0, \Phi \rangle$$

where $Q = Q_s \cup Q_x \cup \{(q, a, q'_x) | q \in Q_x, a \in A, q \overset{a}{\rightarrow} q'_x\}$

**Definition 6.4.** Let $s \in \mathcal{LTS}^S$, and choose $\chi \in \mathcal{LTS}^S$ with traces $(\chi) = L^A_x$ and $Q_x \cap Q_x = \emptyset$. Then $\Xi(s) \in \mathcal{LTS}^S$ is defined by

$$\Xi(s) = \langle Q, \delta, q_0, \Phi \rangle$$

where $Q = Q_s \cup Q_x \cup \{(q, a, q'_x) | q \in Q_x, a \in A, q \overset{a}{\rightarrow} q'_x\}$

The left LTS in Figure 6 shows the result of applying $\Gamma$ and $\Xi$ 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 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 $s \in \mathcal{LTS}^S$. Then

1. $(\Xi(s) = \text{Itraces}(s))$
2. $(\Xi(s) \in \mathcal{LTS}^S$ 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 $ic$ on LTSes. This is done in two steps, and resembles the construction of validation by Beneš et al. [2], as well as transformation $D$ by Bourdonov and Kossatchev [5]. The first step ($\zeta$) ensures quiescence reducibility and stability, while the second ($\eta$) ensures non-blockingness.
Definition 6.6. Let \( s \in \mathcal{LTS}^8 \). Then we define \( \zeta(s) \in \mathcal{LTS}^8 \) by

\[
Q_{\zeta(s)} = \mathcal{P}(Q(s)) \setminus \{q\}
\]

\[
T_{\zeta(s)} = \{ (r, \ell, r \text{after} \ell) \mid r \in Q_{\zeta(s)}, \ell \in A, \forall q \in r : q \Rightarrow \ell \}
\]

\[
\cup \{ (r, \delta, \mathcal{R}(r)) \mid r \in Q_{\zeta(s)}, \forall q \in r, \forall n \in \mathbb{N} : q \Rightarrow \delta^n x_s \}
\]

where \( \mathcal{R}(r) = \{ \mathcal{R}(x_s) \mid n \in \mathbb{N} \} \)

\[
q_{\zeta(s)} = \{ q \}
\]

In the construction of \( \zeta(s) \), a singleton state \( \{q\} \) has the same behaviour as \( q \) in the original LTS \( s \) for actions \( A \). The behaviour for \( \delta \) is changed: \( \delta \)-transitions in \( s \) may cause \( \zeta(s) \) to move from a singleton to a non-singleton state. Intuitively, \( \zeta(s) \) after \( \delta \) is the set of states of \( s \) which are reached by \( \delta \), or by any trace created by removing or duplicating \( \delta \)-occurrences from \( \delta \). 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 \( \zeta \) to \( \Xi(\Gamma(s_d)) \). From the initial state, after \( a \) or \( x \), the behaviour is not changed with respect to \( \Xi(\Gamma(s_d)) \) itself. We find \( \mathcal{R}(\{q_0\}) = \{q_0, q_1\} \), so \( \{q_0\} = \delta \zeta(\Xi(\Gamma(s_d))) = \{q_0, q_1\} \). This is because state \( \delta \) reaches \( q_1 \), but trace \( \epsilon \) (which is a trace with a \( \delta \)-occurrence omitted) reaches \( q_0 \). State \( \{q_0, q_1\} \) is similar to \( q_1 \) in \( \Xi(\Gamma(s_d)) \), except that the latter contains trace \( a \), whereas the former does not, since \( q_0 \). Consequently, unimplementable trace \( \delta a \delta \) is removed by \( \zeta \). This yields a quiescence reducible LTS, trace equivalent to specification \( \Delta(s_c) \).

Example 6.8. Figure 7 shows \( \zeta \) 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 unspecified, so this trace leads to \( q_X \) in \( \Xi(\Gamma(s_d)) \). If \( \delta \)-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, \( q_X \)\}. This state allows only outputs allowed by each of the individual states. All outputs are allowed by \( q_X \), but states 8 and 9 allow respectively \{y\} and \{x\}. The intersection is empty, so \{3, 8, 9, \( q_X \)\} is blocking.

Lemma 6.9. Let \( s \in \mathcal{LTS}^8 \). Then \( \zeta(s) \in \mathcal{LTS}^8 \), and the largest quiescence reducible and stable subset of traces \( s \) is traces \( \zeta(s) \).

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

Definition 6.10. Let \( s \in \mathcal{LTS}^8 \). Then the set of invalid states, denoted by \( \text{inv}(s) \subseteq \mathcal{Q}_s \), is defined as the smallest set of states \( q \in \mathcal{Q}_s \), for which

- \( \exists a \in A_1 : q \text{ after } a \subseteq \text{inv}(s) \), 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 \( \eta(s) \nsucceq \perp \) with traces (\( \perp \))\nsucceq \emptyset .

If \( q_0^s \notin \text{inv}(s) \), then we define \( \eta(s) \in \mathcal{LTS}^8 \) by

\[
Q_{\eta(s)} = Q(s) \setminus \text{inv}(s)
\]

\[
T_{\eta(s)} = T(s) \cap (Q_{\eta(s)} \times A^8 \times Q_{\eta(s)})
\]

\[
q_{\eta(s)}^0 = q_0^s
\]

Note that a blocking state serves as an inductive basis for \( \text{inv} \), as it vacuously satisfies the second condition for being invalid.
Intuitively, \( \eta \) 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 \( \perp \)) are non-blocking, and preserve quiescence stability and reducibility.

**Lemma 6.11.** Let \( s \in \mathcal{LTS}^\delta \). Then
1. if \( q^0 \notin \text{inv}(s) \), 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
\[
(s)_{\text{ioco}} \text{ = traces } (\eta(\Xi(\Gamma(s))))
\]

**Example 6.13.** In Figure 6, \( (\eta(\Xi(\Gamma(s))) \) only has non-blocking states. Consequently, \( \eta \) leaves it unchanged, which means that \( (s)_{\text{ioco}} \) = traces \( (\eta(\Xi(\Gamma(s))) \). Since \( s_{\text{c}} \) is an IOTS, we have \( (s)_{\text{ioco}} \) = \( \text{Straces}(s) \) = traces \( (\Lambda(s)) \). As \( (\eta(\Xi(\Gamma(s))) \) is trace equivalent to \( \Lambda(s) \), we thus have \( (s)_{\text{ioco}} \) = \( (s)_{\text{c}} \). By Theorem 4.5.4, specifications \( s_{\text{c}} \) and \( s_{\text{ioco}} \) are thus proven to be equivalent.

**Example 6.14.** To find the ioco characterization of \( s_{\text{D}} \), we apply \( \eta \) on \( \Xi(\Gamma(s_{\text{D}})) \), in Figure 7. Blocking state \( \{3,8,9,q_{\text{D}}\} \) is invalid, as well as states \( \{1,4,5,10\} \). quiescence stable or reducible, if traces \( (s) \) is, respectively.

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 \( \Gamma \) and \( \zeta \) are exponential, the construction of the LTS-representation has a double exponential upper bound. Since operation \( \zeta \) 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 \( q_{\text{after}} \) of every state \( q \) and action \( \ell \) is finite and computable.\(^1\) Any trace \( \sigma \) may be nonconformal because some extension \( \sigma' \) of unknown length is blocking. In general, the entire state space of the specification must be checked for blocking states, by construction of \( \eta \), to detect nonconformal traces.

---

\(^1\)Without this assumption, incomputability is trivial, as a single step \( q \) after \( \ell \) may already be incomputable.\([3]\) describes a formal definition of computability of after-sets.
We answered which we can now express as language intersection. In that work, this paves the way for using results from lattice theory in the variants of ioco, such as modal ioco [12], symbolic ioco [6] and finite specifications, and undecidability in general. In particular, specification in ioco has unexpected consequences for the ioco transition systems. If quiescence is abandoned in favour of an explicit forbidden outputs or underspecification inputs, then it is forbidden and if we encounter no such forbidden outputs or underspecification inputs, then σ is allowed. 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 σ ∈ (Aδ)∗, and let s ∈ LTS where T s is finitely branching, and Qs, A1 and A2 are enumerable, and where \( \text{init}(q) \) and \( q \) after \( s \) \( t \) are computable for all \( t \in A \) and all \( q \in Qs \). Determining whether \( \sigma \) uioco σ 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. This 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 \( \Delta \) 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 under-characterization 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 \( b^a b^a \) does not fail when testing for specification \( S_A \), and neither does trace \( c \) when testing for \( SD \). 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 ioco, 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**


