Towards Quality of Model-Based Testing in the **ioco** Framework[*]

Michele Volpato and Jan Tretmans

Institute for Computing and Information Sciences
Radboud Universiteit Nijmegen, 6500 GL Nijmegen, NL
{m.volpato, tretmans}@cs.ru.nl

April 26, 2013

Abstract

Since testing is an expensive process, automatic testing with smart test selection has been proposed as a way to reduce such expense. Such a selection of tests can be done using specification coverage functions. Model-based **ioco** theory, however, uses test suites which are not suitable for easy computation of coverage because of interdependence of their test cases. We define a new test suite that overcomes such problems. Using such a test suite we cast the test selection problem to a specification selection problem that aims at transforming the model to which a system under test must conform, in order to reduce the set of test cases. We give a canonical representation for the newly defined test suite.

1 Introduction

Testing is a technique which is used to validate software systems. If it is done manually, it is often laborious and time-consuming, which leads to search for innovative ways to automate it. Model-based testing enables automation in creating test cases, inferring them from a model [14]. The model describes formally how the system should behave. By extracting test cases from the model and executing them against the system, it is possible to check if the system under test is somehow conforming to the model or not.

Although the soundness of an algorithm, or a tool, that generates test cases from a model is relatively easy to establish, its exhaustiveness is not trivial. Beside the possible infiniteness of test cases, they can be so numerous that executing all of them could take ages. The test selection problem deals with the

[*]Supported by STW project 11763 Integrating Testing And Learning of Interface Automata (ITALIA).

impossibility of ensuring exhaustiveness. In fact it aims at providing criteria for selecting test cases in a way that provides high chance of detecting failures. The error detecting capability of a set of test cases that has been built following these criteria can then be measured by a coverage function with the simple idea that the more this set covers the set of all test cases, the more errors it can spot [15]. In this paper we will address issues that arise while considering test selection on the **ioco** framework. In the **ioco** framework the model, called specification, is a labelled transition system with input and output labels and the system under test, also called implementation, is considered to behave as an input-enabled labelled transition system. Test cases are behavior trees constructed from traces of the specification and an implementation conforms to the specification if, for each trace of the specification, the possible outputs after observing that trace in the implementation are also possible after observing that trace in the specification. The **ioco** theory has been extended in many directions, for some examples see [6, 13, 18, 7], and it has been implemented in several model-based testing tools such as TorX [16], JTorX [3], TGV [12], STG [4], TestGen [10], Uppaal-Tron [11], the Agedis Tool set [9], and the commercial tool Axini Test Manager [2].

In the original **ioco** theory, test cases are constructed from s-traces of the specification, i.e. traces of the model obtained from the specification when absence of output is made explicit with a special label δ. In [17] a particular set of traces is introduced, the u-traces set, which deals with input underspecification. We use **uioco** when we test the conformance based on u-traces. The relation **uioco** is to be preferred to **ioco**, even if slightly weaker, because it handles underspecification for input actions.

For non-trivial cases, the set of u-traces is infinite, so we need a method to select (good) traces from this set. Unfortunately computing coverage for the set of u-traces is not easy, because the elements of this set are not independent from each other, i.e. testing for two different u-traces could give the same result on every possible implementation.

Within the **ioco** framework, another way to select traces, to construct test cases, from a specification is testing the conformance of an implementation with respect to a different, weaker specification [15]. The new specification is weaker in the sense that it allows more conforming implementations. A simple adaptation of **uioco** for non input-enabled labelled transition systems is not enough to select such a specification (see section 3).

In this paper we address these two problems (the non-independence of u-traces and the impossibility to use **uioco** for selecting a weaker specification). More specifically, the contributions of the paper are the following: i) the definition of a new set of traces, called r-traces, whose elements are independent, ii) a variation of **ioco**, called **wioco**, based on r-traces, which can be used for selecting specifications for the test selection problem. Moreover we give a canonical representation of r-traces and an algorithm to obtain it from a specification.

Testing conformance on r-traces is equivalent to testing on u-traces, i.e. implementations that are **uioco** conforming to a specification, are also conforming to it w.r.t. r-traces and vice versa. Furthermore r-traces are those u-traces that must be considered to find **uioco** conforming implementations and those only, i.e. it is the smallest set of traces that can be used to check for **uioco** implementation. This extends **ioco** for non input-enabled implementations.

We will give a canonical representation of the r-traces set in form of a *finite state automaton* – if the set of states in the specification is finite – and we will give an algorithm to build this automaton starting from the specification. This is the basis for two test selection methods that can be used to reduce the set of possible tests.

2 Model Formalism and Framework

Model-based testing theory is based on formal and unambiguous models. We model the behavior of systems using *labelled transition systems*. In this section, we recall some definitions directly from [15] regarding labelled transition systems and **ioco** testing theory, and we extend one of them inspired by the work in [6].

A labelled transition system (with input and output labels) is a 5-tuple $\langle Q, L_I, L_U, T, q_0 \rangle$, where Q is a set of states, L_I and L_U are the set of input and output labels or actions, respectively, such that $L_I \cap L_U = \emptyset$, $T \subseteq Q \times (L_I \cup L_U \cup \{\tau\}) \times Q$ is the transition relation and q_0 is the initial state. The special label τ is used to mark internal unobservable transitions. We use the name of the labelled transition system or its initial state interchangeably. We write $q \xrightarrow{\lambda} q'$ rather than $(q, \lambda, q') \in T$, and $\exists q' \in Q : q \xrightarrow{\lambda} q'$ is abbreviated to $q \xrightarrow{\lambda}$. We use L for $L_I \cup L_U$. We denote the class of all labelled transition systems over L_I and L_U by $\mathcal{LTS}(L_I, L_U)$. Let q, q' be states of a labelled transition system $s \in \mathcal{LTS}(L_I, L_U)$, $\lambda \in L$, $\lambda_1 \cdot \ldots \cdot \lambda_n = \sigma \in L^*$ and ϵ be the empty sequence, then we define:

$$
\begin{aligned}
q \overset{\epsilon}{\Rightarrow} q' &\iff q = q' \text{ or } q \xrightarrow{\tau} \ldots \xrightarrow{\tau} q' \\
q \overset{\lambda}{\Rightarrow} q' &\iff \exists q_1, q_2 : q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{\lambda} q_2 \overset{\epsilon}{\Rightarrow} q' \\
q \overset{\sigma}{\Rightarrow} q' &\iff q \overset{\lambda_1}{\Longrightarrow} \ldots \overset{\lambda_n}{\Longrightarrow} q' \\
q \overset{\sigma}{\Rightarrow} &\iff \exists q' \in Q : q \overset{\sigma}{\Rightarrow} q'
\end{aligned}
$$

The set of traces of s is $traces(s) = \{\sigma \in L^* \mid s \overset{\sigma}{\Rightarrow}\}$. Given a trace $\sigma \in L^*$ composed by a sequence of at least k labels, we define $\sigma[k]$ as the k-th label in σ and we define $\sigma^{(k)}$ as the prefix of σ of length k. The length of a trace σ is denoted by $|\sigma|$ and it is identified by the number of visible transitions which compose σ, excluding ϵ- and τ-transitions.

We denote the set of states that are reachable from a given state q via a σ as q **after** $\sigma = \{q' \mid q \overset{\sigma}{\Rightarrow} q'\}$ and by extension, given a set of states P: P **after** $\sigma = \bigcup\{q \text{ **after** } \sigma \mid q \in P\}$.

A set of labels A is refused by a set of states P if $\exists q \in P, \forall \lambda \in A \cup \{\tau\} : q \overset{\lambda}{\not\to}$ and we write it as P **refuses** A. A state q is quiescent if $\{q\}$ **refuses** L_U. Dually, a set of states P must enable a set of labels A if $\forall p \in P, \exists \lambda \in A : p \overset{\lambda}{\Rightarrow}$

and we write it as P **must** A. It is common to assume *strong convergence*, i.e. the system cannot perform an infinite sequence of internal transitions. In case of strong convergence, the following proposition is valid:

Proposition 2.1. *Let* $s \in \mathcal{LTS}(L_I, L_U)$, $\sigma \in L_\delta^*$ *and* $\lambda \in L_I \cup L_U$, *then*

$$(s \textbf{\textit{ after }} \sigma \textbf{\textit{ must }} \{\lambda\}) \iff \neg(s \textbf{\textit{ after }} \sigma \textbf{\textit{ refuses }} \{\lambda\})$$

Let $\delta \notin L_I \cup L_U$ and s_δ be the labelled transition system s to which we add a δ-loop transition $q \xrightarrow{\delta} q$ to all quiescent states, then the set of suspension traces (s-traces) is $Straces(s) = \{\sigma \in L_\delta^* \mid s_\delta \overset{\sigma}{\Rightarrow}\}$, where L_δ^* is the set of sequences of actions in $L \cup \{\delta\}$. From now on we will include δ-transitions in the transition relation, unless otherwise indicated.

We denote the set of outputs, including quiescence, that can be observed from a set of states P as $out(P) = \{\lambda \in L_U \cup \{\delta\} \mid \exists q' \in P : q' \overset{\lambda}{\Rightarrow}\}$.

If $\forall q \in Q, \lambda \in L_I : q \overset{\lambda}{\Rightarrow}$ then we say that the system is input-enabled. The class of all input-enabled labelled transition systems over L_I and L_U is denoted $\mathcal{IOTS}(L_I, L_U)$ and its elements are called *input-output transition systems*.

Given a set $F \subseteq L_\delta^*$, an implementation $i \in \mathcal{IOTS}(L_I, L_U)$ and a specification $s \in \mathcal{LTS}(L_I, L_U)$, we define:

$$\begin{aligned} i \;\mathbf{ioco}_F\; s &\iff \forall \sigma \in F : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma) \\ i \;\mathbf{ioco}\; s &\iff i \;\mathbf{ioco}_{Straces(s)}\; s \\ i \leq_{ior} s &\iff i \;\mathbf{ioco}_{L_\delta^*}\; s. \end{aligned}$$

In [17], another kind of traces, called u-traces, is introduced and studied. It differs from s-traces on so called *underspecified* traces:

$$\begin{aligned} Utraces(s) = \{\sigma \in Straces(s) \mid & \\ & \forall \sigma_1 \in L_\delta^*, \lambda \in L_I : \\ & \sigma_1 \cdot \lambda \preceq \sigma \implies s \textbf{ after } \sigma_1 \textbf{ must } \{\lambda\}\} \end{aligned}$$

where $\preceq$ is the prefix relation. We write i **uioco** s for $i \;\mathbf{ioco}_{Utraces(s)}\; s$.

Lemma 2.2. *Let* $s \in \mathcal{LTS}(L_I, L_U)$ *and* $\lambda \in L_U$,

1. $\sigma \in Utraces(s) \wedge \sigma_1 \preceq \sigma \implies \sigma_1 \in Utraces(s)$
2. $\sigma \in Utraces(s) \wedge \sigma \cdot \lambda \in Straces(s) \implies \sigma \cdot \lambda \in Utraces(s)$

Proof. Immediate from definitions.

□

The relations **ioco** and **uioco** are defined only on input-enabled implementations, but in this paper we consider relations among labelled transition systems. In general, a simple generalization of **uioco** for labelled transition systems is not enough, because, in that case, partial or even trivial implementations result to be conforming to any specification. This is due to the fact that, with such a generalization, implementations are not *forced* to accept inputs that are specified by the specification. Thus we need to consider a variation of **uioco** which handles someway the inputs that are indicated by the specification. The work in [6] inspires a possible solution, also similar to alternating simulations [1], to the non input-enabled specification issue by using the set of inputs after a trace σ, $in(p \textbf{ after } \sigma)$, which are the inputs that a system must be able to execute in all nondeterministically reachable states. This approach adds a new constraint to the definition of **uioco**: the implementation has not only to lead to an output that is foreseen by the specification for each u-trace (i.e. being classical **uioco** conforming), but it also must allow any input that is specified by the specification after each u-trace. Formally: let $s, s' \in \mathcal{LTS}(L_I, L_U)$, $\sigma \in L_\delta^*$, P be a set of states and $\lambda \in L$, then

1. $in(P) = \{\lambda \in L_I \mid P \textbf{ must } \{\lambda\}\}$
2. $s \textbf{ uioco } s' \iff \forall \sigma \in Utraces(s')$:

$$out(s \textbf{ after } \sigma) \subseteq out(s' \textbf{ after } \sigma) \wedge$$
$$in(s' \textbf{ after } \sigma) \subseteq in(s \textbf{ after } \sigma)$$

Note that the new definition of **uioco** coincides with the traditional one in case s is input-enabled.

Lemma 2.3. *Let $s, s' \in \mathcal{LTS}(L_I, L_U)$ and suppose s **uioco** s'. Then*

1. $\sigma \in Straces(s) \wedge \sigma \in Utraces(s') \implies \sigma \in Utraces(s)$
2. $\sigma \notin Straces(s) \wedge \sigma \in Utraces(s') \implies$
   $\exists \sigma_1 \in Straces(s), \lambda \in L_U : \sigma_1 \cdot \lambda \preceq \sigma \wedge \sigma_1 \cdot \lambda \notin Straces(s)$
3. $\sigma \in Straces(s) \wedge \sigma \notin Straces(s') \implies$
   $\exists \sigma_1 \in Utraces(s'), \lambda \in L_I : \sigma_1 \cdot \lambda \preceq \sigma \wedge \sigma_1 \cdot \lambda \notin Utraces(s')$

Proof. 1. Immediate from definitions.

2. Let σ_1 be the longest prefix of σ such that $\sigma_1 \in Straces(s)$. Then $\exists \lambda \in L : \sigma_1 \cdot \lambda \preceq \sigma$ and $\sigma_1 \cdot \lambda \notin Straces(s)$. Suppose $\lambda \in L_I$. By Lemma 2.2.1, $\sigma_1 \cdot \lambda \in Utraces(s')$. Thus, by definition of $Utraces(s')$, $\lambda \in in(s' \textbf{ after } \sigma_1)$. By Lemma 2.2.1, $\sigma_1 \in Utraces(s')$. Thus, since s **uioco** s', $in(s' \textbf{ after } \sigma) \subseteq in(s \textbf{ after } \sigma)$ and so $\lambda \in in(s \textbf{ after } \sigma_1)$. But this implies $\sigma_1 \cdot \lambda \in Straces(s)$ which is a contradiction. Hence $\lambda \in L_U$, as required.

3. Note that $\sigma \notin Utraces(s')$. Let σ_1 be the longest prefix of σ such that $\sigma_1 \in Utraces(s')$. It is trivial that $\sigma_1 \neq \sigma$, thus $\exists \lambda \in L$ such that $\sigma_1 \cdot \lambda \preceq \sigma$ and $\sigma_1 \cdot \lambda \notin Utraces(s')$. We will show that $\lambda \in L_I$. If $\sigma_1 \cdot \lambda \in Straces(s')$ then it is trivial from the definition of $Utraces(s')$. If $\sigma_1 \cdot \lambda \notin Straces(s')$ then $\lambda \notin out(s \textbf{ after } \sigma_1)$. Thus, since s **uioco** s', $out(s \textbf{ after } \sigma_1) \subseteq$

$out(s'$ **after** $\sigma_1)$ and so $\lambda \notin out(s$ **after** $\sigma_1)$. Hence, for the prefix closure of $Straces(s)$, $\lambda \in L_I$ as required.

□

Lemma 2.4. *The relation **uioco** between specifications is reflexive and transitive.*

Proof. Reflexivity is trivial, given the definition of **uioco** between specifications.

For transitivity, suppose that s **uioco** s', s' **uioco** s'' and $\sigma \in Utraces(s'')$. It suffices to show that $out(s$ **after** $\sigma) \subseteq out(s''$ **after** $\sigma) \wedge in(s''$ **after** $\sigma) \subseteq in(s$ **after** $\sigma)$. We consider two cases.

Case 1: $\sigma \notin Straces(s)$. In this case, $out(s$ **after** $\sigma) = \emptyset$ and $in(s$ **after** $\sigma) = L_I$ and the required inclusions follow trivially.

Case2: $\sigma \in Straces(s)$. We claim that also $\sigma \in Straces(s')$. Because suppose that it is not. Then, by Lemma 2.3.3, $\exists \sigma_1 \in Utraces(s'), \lambda \in L_I : \sigma_1 \cdot \lambda \preceq \sigma \wedge \sigma_1 \cdot \lambda \notin Utraces(s')$. Since $\sigma_1 \in Utraces(s'')$ and s' **uioco** s'', $in(s''$ **after** $\sigma_1) \subseteq in(s'$ **after** $\sigma_1)$. Since $\sigma_1 \cdot \lambda \in Utraces(s'')$, $\lambda \in in(s''$ **after** $\sigma_1)$. Hence $\lambda \in in(s'$ **after** $\sigma_1)$ and thus $\sigma_1 \cdot \lambda \in Utraces(s')$, which is a contradiction. Hence $\sigma \in Straces(s')$ and, by Lemma 2.3.1, $\sigma \in Utraces(s')$. Since s **uioco** s'', $out(s$ **after** $\sigma) \subseteq out(s'$ **after** $\sigma)$ and $in(s'$ **after** $\sigma) \subseteq in(s$ **after** $\sigma)$. Since s' **uioco** s''', $out(s'$ **after** $\sigma) \subseteq out(s''$ **after** $\sigma)$ and $in(s''$ **after** $\sigma) \subseteq in(s'$ **after** $\sigma)$. By combining the inclusions we obtain $out(s$ **after** $\sigma) \subseteq out(s''$ **after** $\sigma)$ and $in(s''$ **after** $\sigma) \subseteq in(s$ **after** $\sigma)$, as required.

□

In [19] repetitive quiescence has been addressed with the result that, in a labelled transition system, the behavior *after* observing quiescence is the same if quiescence is observed again. This leads to the following proposition that will be used later in the paper:

Proposition 2.5. *Let $s \in \mathcal{LTS}(L_I, L_U)$, $\sigma_1, \sigma_2 \in L_\delta^*$:*

1. $\sigma_1 \cdot \delta \cdot \delta \cdot \sigma_2 \in Utraces(s) \iff \sigma_1 \cdot \delta \cdot \sigma_2 \in Utraces(s)$
2. $out(s$ ***after*** $\sigma_1 \cdot \delta \cdot \delta \cdot \sigma_2) = out(s$ ***after*** $\sigma_1 \cdot \delta \cdot \sigma_2)$

3 Relate Specifications for Test Selection

Fully testing the conformance of an implementation w.r.t. a model is, in most of the cases, a task that cannot be carried out within a reasonable time. The possible high number of combinations of inputs and the presence of cycles in the specification give a high number (often infinite) of test cases and testing all of them is not feasible. Restricting the set of test cases to those that allow to find failures with a certain (possibly high) probability, is the goal of *test selection*. Many selection techniques have been studied in the domain of conformance testing of labelled transition systems [8], such as heuristic criteria selection [5].

Given a specification s and an implementation under test i, an exhaustive test procedure, to verify if i is conforming to s, would be testing i for all the traces of s. However this could lead to a very long, even infinite, process. Testing i only for a smaller and finite number of traces, which correspond to a specification s' that is weaker than s, and defining a distance between s and s', allows us to say that i is conforming to s with a certain level or measure. Figure 1 explains this idea. $SUT(s)$ is the class of implementations conforming to s: in this paper we focus on **uioco** conformance, thus

$$SUT(s) \equiv \{i \in \mathcal{IOTS}(L_I, L_U) \mid i \textbf{ uioco } s\}$$

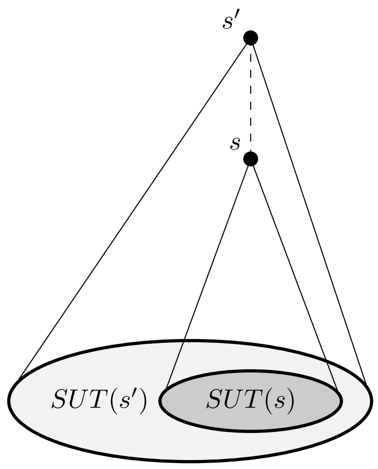
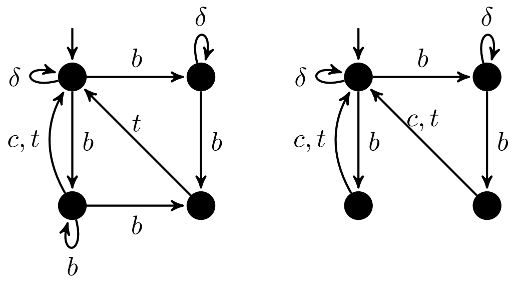


Figure 1: Relation of weakness between specifications: s' is weaker than s because it allows more conforming implementations.

A specification s refines a specification s' if $SUT(s) \subseteq SUT(s')$; this relation leads to the definition of a preorder among specifications. Being able to quantify the distance between a specification and another (comparable) one leads to map the test selection problem to a (distance) minimization problem with a constraint on test cost, e.g. the number of traces to be tested.

Example 3.1. *The specification s of figure 2a describes a vending machine. The only input is b (pressing a button) and the possible outputs are c and t (coffee and tea). The delivery of coffee or tea depends on how many times the button is pressed. If the button is pressed only once either coffee or tea may be delivered. But the machine has also the possibility to remain quiescent, in which case only tea can be delivered (after pressing the button again). If quiescence is not observed and the button is pressed twice or more, the machine delivers coffee or tea, eventually. The specification s' of figure 2b is obtained from s by removing two input transitions and by adding an output transition. It is not difficult to see that if an implementation is **uioco** conforming to s, it is **uioco** conforming also to s': in s' the trace $b{\cdot}b$ is underspecified, this means that any*



(a) Specification s. (b) Specification s'.

Figure 2: Two possible specifications of a vending machine.

behavior is possible after that trace in any ***uioco*** *conforming implementation. Furthermore the sets of output transitions that are enabled in s after $b{\cdot}\delta{\cdot}b{\cdot}t$ and its prefixes and after $b{\cdot}x$ and its prefixes, where x is an output label, are subsets of the ones in s' after the same traces. Thus s' is weaker than s, and an implementation that fails* ***uioco*** *for s' must fail it also for s.*

The goal of this paper is to improve traditional **ioco** theory, focusing on **uioco**, following the qualitative aspect of test selection, laying the basis for a future study of quantitative test selection for this framework. Thus the goal is to define a relation $\mathcal{R}$, defined on labelled transition systems, such that $s\ \mathcal{R}\ s' \iff SUT(s) \subseteq SUT(s')$. The adaptation of **uioco** to not input-enabled implementations does not satisfy this constraint. For instance, even if it is true that when two specifications are **uioco** conforming one w.r.t the other then their sets of **uioco** conforming implementations are in a subset relation (theorem 3.1), the other way around is not always valid. Example 3.2 shows it.

Theorem 3.1. *Let $s, s' \in \mathcal{LTS}(L_I, L_U)$ be two specifications over the same input and output sets,*

$$s\ \textbf{\textit{uioco}}\ s' \implies SUT(s) \subseteq SUT(s')$$

Proof. Assume s **uioco** s' and suppose $s'' \in SUT(s)$. It suffices to prove $s'' \in SUT(s')$. By the definition of $SUT(s)$, s'' **uioco** s and because of the transitivity of **uioco**, s'' **uioco** s'. Thus $s'' \in SUT(s')$.

□

Example 3.2. *The specification s'' of figure 3 is obtained from the specification s' of figure 2b by making explicit the underspecification of trace $b{\cdot}b$. The fact that we have only made explicit the underspecification implies that $SUT(s') = SUT(s'')$, but s' $\textbf{\textit{ui}}\not{o}\textbf{\textit{co}}$ s'', because:*

$$b \in Utraces(s''),\ \text{but}\ \{b\} = in(s''\ \textbf{\textit{after}}\ b) \not\subseteq in(s'\ \textbf{\textit{after}}\ b) = \emptyset.$$

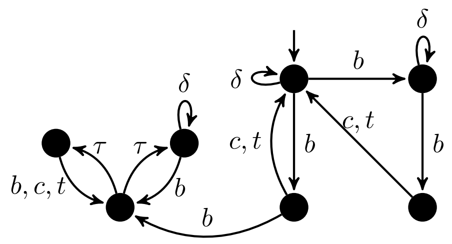


Figure 3: Specification s''. It has the same **uioco** conforming implementations of s' in figure 2b, but s' $\mathbf{ui}\not{o}\mathbf{co}$ s''.

In the next section we give a way to relate labelled transition systems which satisfies the constrains of the relation $\mathcal{R}$. It is based on the input-output transition system that is obtained from a specification by applying *demonic completion.*

3.1 Demonic Completion

Example 3.2 suggests a possible approach we can follow to obtain a successful relation between specifications: we can make the concept of underspecified trace explicit, adding transitions and states to a specification in order to complete it. The final complete specification must have the same behavior of the initial one from the **uioco** conformance point of view, i.e. the sets of **uioco** conforming implementations must be the same.

Demonic completion [17] goes well for this purpose. It introduces a *chaotic state*, a state that accepts any input and may generate any output.

Definition 3.1 (Chaotic state χ). *A chaotic state χ is a state with the following properties:*

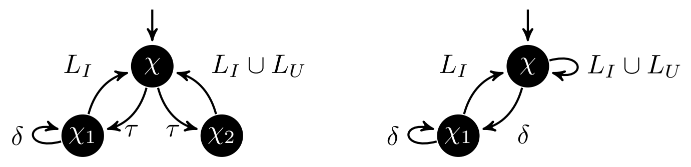
- $\forall \lambda \in L_I \cup L_U : \chi \overset{\lambda}{\Rightarrow} \chi$
- $\forall a \in L_I : \chi \overset{\delta \cdot a}{\Longrightarrow} \chi$

where L_I and L_U are the sets of input and output labels respectively and δ represents quiescence.

A chaotic state can be constructed in many ways, figure 4 shows a nondeterministic and a deterministic one.

The *demonic completion* of a specification is the input-output transition system obtained by adding a chaotic state χ to that specification such that all non-specified inputs lead to χ. In the following definition the deterministic version of a chaotic state (figure 4b) is used.

Definition 3.2. *Given a labelled transition system $s = \langle Q, L_I, L_U, T, q_0 \rangle$, the*



(a) Nondeterministic chaotic state χ.

(b) Deterministic chaotic state χ.

Figure 4: Graphical representations of *chaotic behavior*.

demonic completion $DC(s)$ *of* s *is a tuple* $\langle Q', L_I, L_U, T', q_0 \rangle$ *where:*

$$\begin{aligned} Q' &= Q \cup \{\chi, \chi_1\} \text{ with } \chi, \chi_1 \notin Q \\ T' &= T \cup \{(q, a, \chi) \mid q \in Q, a \in L_I, q \overset{a}{\not\to}, q \overset{\tau}{\not\to}\} \\ &\quad \cup \{(\chi, \delta, \chi_1)\} \\ &\quad \cup \{(\chi_1, a, \chi) \mid a \in L_I\} \\ &\quad \cup \{(\chi, \lambda, \chi) \mid \lambda \in L_I \cup L_U\} \end{aligned}$$

The demonic completion of a labelled transition system is an input-output transition system, i.e. it is input-enabled. Moreover once the state χ is reached, any input or (absence of) output is accepted. This is exactly the meaning we give to underspecified inputs in a specification: if an input is not specified, we do not care how the implementation acts after that input action. These considerations lead to the following propositions:

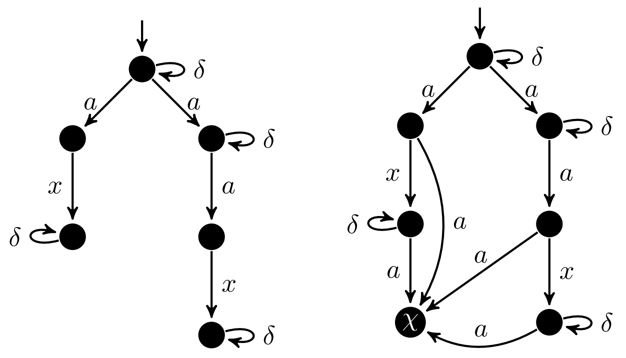
Proposition 3.2. *Let* $s, s' \in \mathcal{LTS}(L_I, L_U)$ *be two specifications, and* $DC(s), DC(s')$ *be their demonic completions,*

1. $DC(s), DC(s') \in \mathcal{IOTS}(L_I, L_U)$ *and:*
   (a) $DC(s)$ ***uioco*** $DC(s') \iff DC(s)$ ***ioco*** $DC(s')$
   (b) $DC(s)$ ***uioco*** $DC(s') \iff \mathit{Straces}(DC(s)) \subseteq \mathit{Straces}(DC(s'))$
2. $DC(s)$ ***uioco*** s
3. $\sigma \in \mathit{Straces}(s) \implies \sigma \in \mathit{Straces}(DC(s))$

Note that, if a specification s is nondeterministic, then $DC(s)$ **ioco** s does not always hold, in fact for the **ioco** relation an input after a trace σ is underspecified only if all the states reachable after σ do not enable that input. We can see an example in figure 5.

In this figure the trace $a{\cdot}a$ is not ioco-underspecified in s because there exists a state reachable from s after $a{\cdot}a$, thus its demonic completion $DC(s)$ is not **ioco** related to s. More formally, for $\sigma = a{\cdot}a \in \mathit{Straces}(s)$:

$$\{x, \delta\} = \mathit{out}(DC(s) \textbf{ after } \sigma) \not\subseteq \mathit{out}(s \textbf{ after } \sigma) = \{x\}$$



(a) Specification s.

(b) Demonic completion $DC(s)$.

Figure 5: A specification $s \in \mathcal{LTS}(\{a\}, \{x\})$ and its demonic completion. State χ_1 is not represented for the sake of clarity.

In fact, $DC(s)$ **after** $a{\cdot}a$ contains the chaotic state χ that allows any output and quiescence.

We made input underspecifications explicit through demonic completion. Now we can relate the demonic completions of two specifications. Quiescent trace preorder $\leq_{ior}$ on demonically completed specifications preserves a preorder also on the sets of **uioco** conforming implementations (theorem 3.4).

First we present a lemma that will be used in the proof of the main theorem. The following lemma claims that $Straces(DC(s))$ contains all the s-traces that any implementation $i \in SUT(s)$ could produce.

Lemma 3.3. *Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification and $DC(s)$ be its demonic completion and let $i \in \mathcal{IOTS}(L_I, L_U)$ be an input-output transition system then*

$$i \ \textbf{\textit{uioco}}\ s \implies i \leq_{ior} DC(s)$$

Proof. We demonstrate an equivalent statement:

$$i \ \textbf{uioco}\ s \implies Straces(i) \subseteq Straces(DC(s))$$

Let us consider an arbitrary trace $\sigma \in Straces(i)$. We split the proof into two cases. First we consider the case in which $\sigma \in Straces(s)$:

$$\sigma \in Straces(s) \implies \sigma \in Straces(DC(s)) \qquad \text{(for preposition 3.2.3)}$$

In the second case, if $\sigma \notin Straces(s)$ then, for lemma 2.3.3,

$$\exists \sigma_1 \in Utraces(s), \lambda \in L_I : \sigma_1{\cdot}\lambda \preceq \sigma \wedge \sigma_1{\cdot}\lambda \notin Utraces(s)$$

which means that

$$\begin{aligned}
& s \textbf{ after } \sigma_1 \textbf{ refuses } \{\lambda\} && \text{(definition of } \mathit{Utraces}(s) \text{ and proposition 2.1)}\\
& \implies \chi \in (DC(s) \textbf{ after } \sigma_1{\cdot}\lambda) && \text{(for the definition of } DC(s))\\
& \implies \sigma \in \mathit{Straces}(DC(s)) && \text{(for the definition of } \chi)
\end{aligned}$$

Thus, in both cases, $\sigma \in \mathit{Straces}(DC(s))$, so $\mathit{Straces}(i) \subseteq \mathit{Straces}(DC(s))$, which implies $i \leq_{ior} DC(s)$.

□

Theorem 3.4. *Let $s, s' \in \mathcal{LTS}(L_I, L_U)$ be two specifications over the same input and output sets,*

$$SUT(s) \subseteq SUT(s') \iff DC(s) \leq_{ior} DC(s')$$

Proof. ($\Rightarrow$)

$$\begin{aligned}
& SUT(s) \subseteq SUT(s') \wedge \text{preposition 3.2.2}\\
& \implies DC(s) \in SUT(s')\\
& \implies DC(s) \leq_{ior} DC(s') && \text{(for lemma 3.3)}
\end{aligned}$$

($\Leftarrow$)
We prove the contrapositive:

$$SUT(s) \nsubseteq SUT(s') \implies DC(s) \nleq_{ior} DC(s')$$

So that

$$SUT(s) \nsubseteq SUT(s') \implies \mathit{Straces}(DC(s)) \nsubseteq \mathit{Straces}(DC(s'))$$

$$\begin{aligned}
& SUT(s) \nsubseteq SUT(s')\\
& \quad \implies \exists i \in \mathcal{IOTS}(L_I, L_U) : i \textbf{ uioco } s \wedge i \textbf{ ui}\not{\textbf{o}}\textbf{co } s'\\
& \quad \implies \exists i, \exists \sigma \in \mathit{Utraces}(s') : \mathit{out}(i \textbf{ after } \sigma) \nsubseteq \mathit{out}(s' \textbf{ after } \sigma)\\
& \qquad \wedge\ (i \textbf{ uioco } s) && (1)\\
& \quad \implies \exists i, \exists \sigma, \exists \lambda \in L_U : \lambda \in \mathit{out}(i \textbf{ after } \sigma) \wedge \lambda \notin \mathit{out}(s' \textbf{ after } \sigma) && (2)
\end{aligned}$$

Let i be the implementation, σ be the trace and λ be the output label that satisfy (2):

$$\begin{aligned}
(2) & \implies \sigma{\cdot}\lambda \notin \mathit{Straces}(s') \wedge \sigma{\cdot}\lambda \in \mathit{Straces}(i)\\
& \qquad \text{(and for proposition 3.2.2, lemma 3.3 and for (1))}\\
& \implies \sigma{\cdot}\lambda \notin \mathit{Straces}(DC(s')) \wedge \sigma{\cdot}\lambda \in \mathit{Straces}(DC(s))\\
& \implies \mathit{Straces}(DC(s)) \nsubseteq \mathit{Straces}(DC(s'))
\end{aligned}$$

as required.

□

In conclusion, comparing the s-traces of the demonic completions of two specifications allows us to infer a relation between the set of **uioco** conforming implementations of these specifications.

4 The Set of Required Traces

In the previous section we proposed a way to relate demonically completed specifications in order to reason about the relations between their **uioco** conforming implementations, but these results involve the computation of demonic completions and the comparison between traces of such input-output transition systems. A more interesting outcome would be finding a relation among specifications based on a set of their traces such as **ioco** and **uioco**. The set of u-traces could be a candidate, but we can see from previous sections that it contains too many traces. For instance, consider specification s' of figure 2b and s'' of figure 3. It is easy to see that their demonic completions are trace equivalent. Thus, for theorem 3.4, an implementation that is **uioco** conforming to one of them, is **uioco** conforming also to the other. Their set of u-traces, on the contrary, are different. For example the trace $b{\cdot}b$ belongs to $Utraces(s')$ but not to $Utraces(s)$ and we can conclude that $b{\cdot}b$ is not necessary for **uioco** testing. Thus, first, we will reduce the set of u-traces, and then we will define a relation on labelled transition systems which is based on the newly defined set.

In this section, we present the set of required traces. Required traces are, basically, the only traces that are needed for testing **uioco**. The idea behind required traces is removing from u-traces the traces which give same results on all implementations, i.e. those that are testing dependent from other traces. A particular kind of such traces are those traces that are unable to specify any implementation, i.e. those traces for which there does not exist any implementation which can fail **uioco** after them. Example 4.1 presents a notion of these kinds of traces.

Example 4.1. *Consider the specification s of figure 2a. The trace b and the trace $b{\cdot}\delta{\cdot}\delta{\cdot}b$ are both u-traces of s, but they are not needed for **uioco** testing. In fact $out(s$ **after** $b) = \{c, t, \delta\}$ which is the entire set of output labels plus δ, so testing after b is useless. Moreover, given proposition 2.5, for any input-output transition system i, $out(i$ **after** $b{\cdot}\delta{\cdot}\delta{\cdot}b) =$
$out(i$ **after** $b{\cdot}\delta{\cdot}b)$, and thus it is enough to test **uioco** only for one of them.*

From proposition 2.5, it is clear that it is not necessary to test all the u-traces of a specification s to check for **uioco**. In fact we can avoid u-traces with subsequences of δ transitions, because the result for those u-traces will be the same as for the u-trace obtained by collapsing the subsequences of consecutive δ transitions to a single one.

Avoiding traces containing sequences of δ is not enough, in fact we have to deal also with those traces that are unable to specify any implementation. We can identify such traces by observing the ending label or the set of outputs that is enabled after the trace: if the trace ends with δ then no implementation can accept an output different than δ after that trace (proposition 2.5.2); if the trace enables the entire set L_U and the special label δ, then, again, no implementation can fail **uioco** after it. This reasoning is formally presented in lemma 4.1.

Lemma 4.1. *Let $s \in \mathcal{LTS}(L_I, L_U)$ and $\sigma \in Utraces(s)$:*

$$\sigma \textit{ does not end with } \delta \wedge out(s \textbf{ \textit{after} } \sigma) \neq L_U \cup \{\delta\}$$
$$\Longleftrightarrow$$
$$\exists i \in \mathcal{IOTS}(L_I, L_U) : out(i \textbf{ \textit{after} } \sigma) \nsubseteq out(s \textbf{ \textit{after} } \sigma)$$

Proof. $(\Rightarrow)$

We can build an input-output transition system i such that $out(i \textbf{ after } \sigma) \nsubseteq out(s \textbf{ after } \sigma)$ as follow. Let $\lambda \in (L_U \cup \{\delta\}) \setminus out(s \textbf{ after } \sigma)$. Let us consider $DC_1(s)$ as the demonic completion $DC(s)$ to which we add:

- a new quiescent state ψ,
- transitions (p, λ, ψ) where $p \in (DC(s) \textbf{ after } \sigma)$,
- transitions (ψ, a, χ) where $a \in L_I$.

Figure 6 gives an example of the construction.

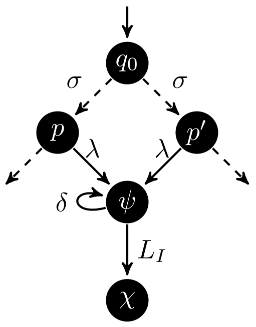


Figure 6: Graphical representation of $DC_1(q_0)$ *construction.*

Let $p \in (DC_1(s) \textbf{ after } \sigma)$, ψ will be the only state that is reachable from p with a transition labelled with λ, in fact no other transitions from p with label λ are present, for proposition 3.2.2 given that $\sigma \in Utraces(s)$ and $\sigma{\cdot}\lambda \notin Straces(s)$.

Note that we can add such elements, maintaining the characteristics of labelled transition systems, because σ does not end with δ [19]. Furthermore ψ is input-enabled, like all other states of $DC_1(s)$ so $DC_1(s) \in \mathcal{IOTS}(L_I, L_U)$. It is obvious that $\lambda \in out(DC_1(s) \textbf{ after } \sigma)$ and $\lambda \notin out(s \textbf{ after } \sigma)$, thus

$$out(DC_1(s) \textbf{ after } \sigma) \nsubseteq out(s \textbf{ after } \sigma)$$

$(\Leftarrow)$
It is trivial that:

$$\exists i \in \mathcal{IOTS}(L_I, L_U) : out(i \textbf{ after } \sigma) \nsubseteq out(s \textbf{ after } \sigma)$$
$$\Longrightarrow out(s \textbf{ after } \sigma) \neq L_U \cup \{\delta\}$$

By contradiction let us assume that σ ends with δ. In this case, in order for s and i to be labelled transition systems, the following must be true:

$$\begin{aligned} out(s \textbf{ after } \sigma) &= \{\delta\} && (\sigma \in Straces(s) \text{ and } [19]) \\ out(i \textbf{ after } \sigma) &= \{\delta\} \vee \emptyset && (\text{see } [19]) \end{aligned}$$

In other words, after quiescence (δ) no other output can be observed until a new input is provided [19]. Thus $out(i \textbf{ after } \sigma)$ is always contained in $out(s \textbf{ after } \sigma)$. Contradiction.

$\square$

The removal of such traces leads to the following definition:

Definition 4.1. *Let $s \in \mathcal{LTS}(L_I, L_U)$. Then*

$$\begin{aligned} Rtraces(s) = \{\sigma \in Utraces(s) \mid & \\ & \delta{\cdot}\delta \textit{ is not a substring of } \sigma \\ & \wedge \sigma \textit{ does not end with } \delta \\ & \wedge out(s \textit{\textbf{ after }} \sigma) \neq L_U \cup \{\delta\}\} \end{aligned}$$

We call an element of $Rtraces(s)$ an *r-trace* of s.

Example 4.2. *Consider the specification s' of figure 2b. The trace b is not an r-trace because $out(s'$ **after** $b) = \{c, t, \delta\}$*
$= L_U \cup \{\delta\}$. Neither $b{\cdot}\delta{\cdot}\delta{\cdot}b$ nor δ are r-traces, because the former contains $\delta{\cdot}\delta$, while the latter ends with δ. All three of them are u-traces of s'.

Being the set of r-traces a subset of the set of u-traces, soundness, which has been proven holding for u-traces, is preserved. The set $Rtraces(s)$ could be a proper subset of $Utraces(s)$ and removing traces from the set of u-traces could weaken the specification, i.e. it could allow more conforming implementations. Theorem 4.2 proves that this is not the case for $Rtraces(s)$.

Theorem 4.2. *Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification and $i \in \mathcal{IOTS}(L_I, L_U)$ be an input-output transition system. Then: i **uioco** $s \iff i$ **ioco**$_{Rtraces(s)}$ s.*

Proof. ($\Rightarrow$)
It is trivial given that $Rtraces(s) \subseteq Utraces(s)$.

($\Leftarrow$)
Given i **ioco**$_{Rtraces(s)}$ s we will demonstrate that:

$$\forall \sigma \in Utraces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

Let $\sigma \in Utraces(s)$ be an arbitrary trace.
If $\sigma \in Rtraces(s)$ then, for the definition of **ioco**$_{Rtraces(s)}$:

$$out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

Otherwise $\sigma \in (Utraces(s) \setminus Rtraces(s))$ and, for definition 4.1:

$$\begin{aligned} & \delta{\cdot}\delta \text{ is a substring of } \sigma && (3)\\ \vee\ & \sigma \text{ ends with } \delta && (4)\\ \vee\ & out(s \textbf{ after } \sigma) = L_U \cup \{\delta\} && (5)\end{aligned}$$

Let us first handle (4) $\vee$ (5). In this case, for lemma 4.1:

$$\forall j \in \mathcal{IOTS}(L_I, L_U) : out(j \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

and in particular for $j = i$:

$$out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$$

On the contrary, if $\neg(4) \wedge \neg(5)$, then (3) must hold. In this case there exists a $\sigma' \in Rtraces(s)$ obtained from σ by reducing all sequences of δ into a single δ, so:

$$\begin{aligned} & out(i \textbf{ after } \sigma') \subseteq out(s \textbf{ after } \sigma') && (6)\\ & (6) \wedge \text{ proposition 2.5.1 } \implies out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma) \end{aligned}$$

□

After reducing the set of u-traces to the set of r-traces, it is not possible to find a set $A \subsetneq Rtraces(s)$ for which it is valid that i **uioco** $s \iff i$ $\textbf{ioco}_A$ s, i.e. all $\sigma \in Rtraces(s)$ are necessary for **uioco** checking. Thus r-traces are, in a particular sense, independent of one another. Formally:

Theorem 4.3. *Let $s \in \mathcal{LTS}(L_I, L_U)$ and $A \subsetneq Rtraces(s)$. Then $\exists i \in \mathcal{IOTS}(L_I, L_U)$: i $\textbf{ioco}_A$ $s \wedge i$ $\textbf{ui}\not\textbf{o}\textbf{co}$ s.*

Proof. We prove the theorem by constructing an input-output transition system i such that, given two arbitrary r-trace σ and σ' such that $\sigma \neq \sigma'$:

$$out(i \textbf{ after } \sigma) \not\subseteq out(s \textbf{ after } \sigma) \wedge out(i \textbf{ after } \sigma') \subseteq out(s \textbf{ after } \sigma').$$

We build such a input-output transition system by following these steps:

1. Compute $DC(s) = \langle Q, L_I, L_U, T, q_0\rangle$ and add δ-loops to quiescent states.
2. Add a new state q_1 to Q and a new transition (q_0, λ, q_1) to T where $\lambda = \sigma[1]$. Then if $\lambda = \delta$ add a δ-loop to q_1.
3. Let $k = 2$, $\sigma_1 = \sigma^{(1)}$ and $\lambda = \sigma[k]$:
    - add a new state q_k to Q and a new transition (q_{k-1}, λ, q_k) to T. Then for all $a \in L_I \mid a \neq \lambda$ add (q_{k-1}, a, p) to T where $p \in (DC(s) \textbf{ after } \sigma_1{\cdot}a)$;
    - if $\lambda \in L_I$ then q_{k-1} is quiescent: add a δ-loop to q_{k-1};

- if $\lambda = \delta$ then add a δ-loop to q_k.

4. Repeat the substeps from step 3 iterating on k replacing σ_1 with $\sigma^{(k)}$ until $k = |\sigma|$. Then for all $a \in L_I$ add $(q_{|\sigma|}, a, p)$ to T where $p \in (DC(s)$ **after** $\sigma{\cdot}a)$;

5. Add a new state r to Q and a transition $(q_{|\sigma|}, x, r)$ to T where $x \in (L_U \cup \{\delta\}) \setminus (out(s$ **after** $\sigma))$ then add a δ-loop to r. Note that we can add any output transition because σ does not end with δ. At the end demonically complete r.

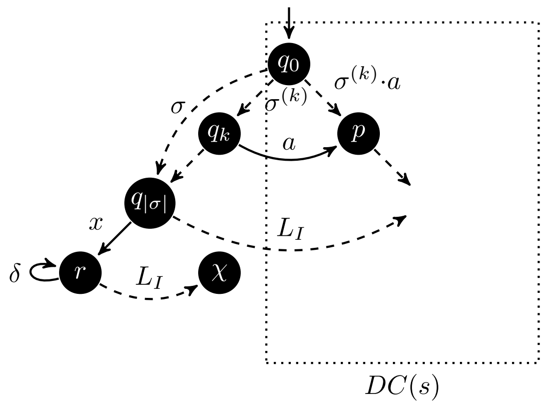


Figure 7: Graphical representation of i *construction.* $\sigma^{(k)}{\cdot}a$ is not a prefix of σ.

Figure 7 shows the result of applying these steps to a generic specification s with an emphasis on iteration k.

The way i has been built makes it a input-output transition system and the last added transition $(q_{|\sigma|}, x, r)$ differs i from $DC(s)$ in terms of traces: $\sigma{\cdot}x$ is a trace of i, but it is not a trace of $DC(s)$ or of any other implementation j such that j **uioco** s (for lemma 3.3). For this reason $out(i$ **after** $\sigma) \not\subseteq out(s$ **after** $\sigma)$. It is trivial that $\sigma' \neq \sigma{\cdot}x$. Furthermore being $\sigma' \neq \sigma$ and the fact that σ' does not contain sequences of δ labels (that would be the only way for σ' to reach q_n) imply that $q_n, r \notin (i$ **after** $\sigma')$ thus for σ' the implementation i behaves as $DC(s)$ that is **uioco** conforming to s. Thus $out(i$ **after** $\sigma') \subseteq out(s$ **after** $\sigma')$. □

Theorem 4.3 is not valid for traditional s-traces or u-traces in general. For example if we consider the specification s of figure 5a, the traces $\delta{\cdot}a{\cdot}x$ and $\delta{\cdot}\delta{\cdot}a{\cdot}x$ are u-traces (and so they are s-traces as well), but the second is not an r-trace. It is also not possible to find a valid implementation i such that

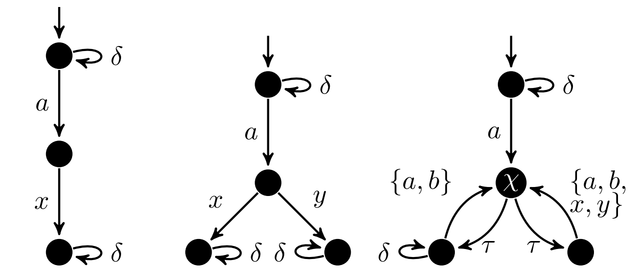
$$out(i \textbf{ after } \delta{\cdot}a{\cdot}x) \not\subseteq out(s \textbf{ after } \delta{\cdot}a{\cdot}x) \wedge$$
$$out(i \textbf{ after } \delta{\cdot}\delta{\cdot}a{\cdot}x) \subseteq out(s \textbf{ after } \delta{\cdot}\delta{\cdot}a{\cdot}x)$$

because in a labelled transition system, the behavior that can be observed after a δ-transition must be observable also after another δ-transition [19].

4.1 Comparing Sets of Required Traces

It would be natural to guess if comparing sets of r-traces could be linked to comparing sets of **uioco** conforming implementations. Unfortunately r-traces do not carry any information on the enabled output after them, so that is not enough. The following example explains it better.

Example 4.3. *Consider the specifications of figure 8 with input and output labels $L_I = \{a, b\}$ and $L_U = \{x, y\}$. Since s* specifies *more than s', i.e. after the trace a it is possible to observe only the output x, while in s' it is possible to observe also the output y, the set of implementations which are **uioco** (or $\textbf{ioco}_{Rtraces(s)}$) conforming to s is a subset of the set of implementations which are **uioco** conforming to s'. The same reasoning can be done for s' and s''. Formally: $SUT(s) \subseteq SUT(s') \subseteq SUT(s'')$.*



(a) Specification s. (b) Specification s'. (c) Specification s''.

Figure 8: Three specifications whose sets of **uioco** conforming implementations are in a subset relation. Their sets of r-traces are not in the same kind of relation.

We can easily find the sets of r-traces:

$$\begin{aligned} Rtraces(s) =& \{\epsilon, \delta\cdot a, a, a\cdot x, \delta\cdot a\cdot x\} \\ Rtraces(s') =& \{\epsilon, \delta\cdot a, a, a\cdot x, \delta\cdot a\cdot x, a\cdot y, \delta\cdot a\cdot y\} \\ Rtraces(s'') =& \{\epsilon\} \end{aligned}$$

and so: $Rtraces(s'') \subseteq Rtraces(s) \subseteq Rtraces(s')$.

From this example it is clear that we need more information in order to find a relation among specifications that is linked to their **uioco** conforming implementations, such as a kind of **uioco** relation between specifications based on r-traces. In the next section we will define such a relation and we will show that it has the properties that we described in section 3.

4.2 Input-output Conformance between Specifications

The set of r-traces can be used to define a relation between specifications which satisfies the constraint we set for $\mathcal{R}$ at the end of section 3, but $\mathbf{ioco}_{Rtraces(s)}$ is defined only between input-output transition systems (implementations) and labelled transition systems (specifications). This is due to the fact that the system we want to check for conformance is *input-enabled*. As for **uioco** between specifications, we need to consider a constraint on the set of inputs that are allowed after each trace, but we are not interested in all of them. Only those input transitions which may extend the current trace to an r-trace must be considered, because if a trace σ concatenated with an input label λ cannot be extended to an r-trace, then we know that there are no implementations which can fail **uioco** after any extension of $\sigma\cdot\lambda$ and thus any behavior is allowed. So we define $Rin(s,\sigma)$ which is stricter than $in(s \textbf{ after } \sigma)$.

Definition 4.2. *Let $s \in \mathcal{LTS}(L_I, L_U)$ be a specification and $\sigma \in L_\delta^*$ be a trace,*

$$Rin(s,\sigma) = \{\lambda \in in(s \textbf{ after } \sigma) \mid \exists \sigma' \in Rtraces(s) : \sigma\cdot\lambda \preceq \sigma'\}$$

The set $Rin(s,\sigma)$ contains those input labels that must be enabled in any conforming implementation after observing σ, in order to ensure that the implementation is able to perform any input action that is specified by s. Such input labels are only those which lead to an r-trace. The adaptation of $\mathbf{ioco}_{Rtraces(s)}$to non-input enabled labelled transition systems can now be defined. In the comparison of sets of inputs that must be enabled, it has to take into account also the prefixes of the r-traces, because $Rtraces(s)$ is not prefix closed.

Definition 4.3. $s \textbf{ wioco } s' \iff \forall \sigma \in Rtraces(s')$:

1. $out(s \textbf{ after } \sigma) \subseteq out(s' \textbf{ after } \sigma) \wedge$
2. $\forall \sigma_1 \preceq \sigma : Rin(s', \sigma_1) \subseteq in(s \textbf{ after } \sigma_1)$

Example 4.4. *Consider s and s' of figure 2. Basically s' is obtained from s by removing two input transitions and adding an output one. The removal of the input transitions made a trace becoming underspecified ($b\cdot b$), and the addition of the output transition, instead, specified a trace that was not ($b\cdot\delta\cdot b\cdot c$), so we added elements to the $out(\dots)$ set and we removed elements from $in(\dots)$ and $Rin(\dots)$ sets. We obtained a weaker specification: $s \textbf{ wioco } s'$.*

As for **uioco** between specifications in the previous section, **wioco** extends $\mathbf{ioco}_{Rtraces(s)}$ to non input-enabled implementations. Note that, in case s is input-enabled, this definition coincides with $s \ \mathbf{ioco}_{Rtraces(s)} \ s'$ and thus, for theorem 4.2, $i \textbf{ wioco } s \iff i \textbf{ uioco } s$ with $i \in \mathcal{IOTS}(L_I, L_U)$.

Lemma 4.4. *Let $s, s' \in \mathcal{LTS}(L_I, L_U)$ and suppose $s \textbf{ wioco } s'$. Then*

$$\sigma \in Rtraces(s') \wedge \sigma \in Straces(s) \implies \sigma \in Utraces(s)$$

Proof. Suppose $\sigma \notin Utraces(s)$ then, for the definition of $Utraces(s)$ and proposition 2.1, $\exists \sigma_1 \cdot \lambda \preceq \sigma, \lambda \in L_I : s$ **after** σ_1 **refuses** $\{\lambda\}$ and so $\lambda \notin in(s$ **after** $\sigma_1)$. On the other hand, $\sigma \in Rtraces(s')$ implies $\lambda \in Rin(s', \sigma_1)$ which is in contradiction with s **wioco** s'.

□

Theorem 4.5 proves that **wioco** satisfies the constrains of relation $\mathcal{R}$ we introduced in section 3.

Theorem 4.5. *Let $s, s' \in \mathcal{LTS}(L_I, L_U)$ be two specifications over the same input and output sets,*

$$s \textbf{\textit{ wioco }} s' \implies SUT(s) \subseteq SUT(s')$$

Proof. We prove an equivalent statement:

$$s \textbf{ wioco } s' \implies DC(s) \textbf{ ioco}_{Rtraces(s')}\, s'$$

which, due to lemma 3.3, proposition 3.2 and the transitivity of **uioco**, implies $SUT(s) \subseteq SUT(s')$.

Given $\sigma \in Rtraces(s')$ we show that $out(DC(s)$ **after** $\sigma) \subseteq out(s'$ **after** $\sigma)$. We consider two cases.

Case1: $\sigma \notin Straces(DC(s))$, then $out(DC(s)$ **after** $\sigma) = \emptyset$ and the inclusion follows trivially.

Case2: $\sigma \in Straces(DC(s))$. We claim that also $\sigma \in Straces(s)$. Because suppose that it is not. Then, by Lemma 2.3.3,

$$\begin{aligned} \exists \sigma_1 \in Utraces(s), \lambda \in L_I : \sigma_1 \cdot \lambda \preceq \sigma \wedge \sigma_1 \cdot \lambda \notin Utraces(s) \\ \implies \lambda \notin in(s \textbf{ after } \sigma_1) \qquad & (7) \\ \sigma \in Rtraces(s') \wedge \sigma_1 \cdot \lambda \preceq \sigma \implies \lambda \in Rin(s', \sigma_1) \qquad & (8) \\ (7) \wedge (8) \implies s \textbf{ wi}\not{\textbf{o}}\textbf{co } s' \qquad & \text{(contradiction)} \end{aligned}$$

So $\sigma \in Straces(s)$. Then for lemma 4.4 $\sigma \in Utraces(s)$ and then for proposition 3.2.2:

$$out(DC(s) \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma) \subseteq out(s' \textbf{ after } \sigma)$$

as required.

□

Theorem 4.6. *Let $s, s' \in \mathcal{LTS}(L_I, L_U)$ be two specifications over the same input and output sets,*

$$SUT(s) \subseteq SUT(s') \implies s \textbf{\textit{ wioco }} s'$$

Proof. We prove the contrapositive:

$$s \textbf{ wi}\not{\textbf{o}}\textbf{co } s' \implies SUT(s) \not\subseteq SUT(s')$$

more specifically we demonstrate (for proposition 3.2.2) that

$$s \ \mathbf{wi\not{o}co} \ s' \implies DC(s) \notin SUT(s')$$

$$\begin{aligned} s \ \mathbf{wi\not{o}co} \ s' \implies & \exists \sigma \in Rtraces(s') : \\ & out(s \ \mathbf{after} \ \sigma) \not\subseteq out(s' \ \mathbf{after} \ \sigma) & (9) \\ & \vee (\exists \sigma_1 \preceq \sigma : Rin(s', \sigma_1) \not\subseteq in(s \ \mathbf{after} \ \sigma_1)) & (10) \end{aligned}$$

Let σ be the r-trace of s' that satisfies the previous statement.

Consider (9) and let $\lambda \in L_U \cup \{\delta\}$ be the output label such that:

$$\lambda \in out(s \ \mathbf{after} \ \sigma) \wedge \lambda \notin out(s' \ \mathbf{after} \ \sigma)$$

this implies $\sigma{\cdot}\lambda \in Straces(s)$.

$$\begin{aligned} \sigma{\cdot}\lambda \in Straces(s) & \implies \sigma{\cdot}\lambda \in Straces(DC(s)) \qquad \text{(for proposition 3.2.3)} \\ & \implies out(DC(s) \ \mathbf{after} \ \sigma) \not\subseteq out(s' \ \mathbf{after} \ \sigma) \\ & \implies \neg(DC(s) \ \mathbf{ioco}_{Rtraces(s)} \ s') \\ & \implies DC(s) \notin SUT(s') \end{aligned}$$

Now let σ_1 be the trace that satisfies (10) and let $\lambda \in L_I$ be the input label such that:

$$\lambda \notin in(s \ \mathbf{after} \ \sigma_1) \wedge \lambda \in Rin(s', \sigma_1)$$

$$\begin{aligned} \lambda \notin in(s \ \mathbf{after} \ \sigma_1) & \implies s \ \mathbf{after} \ \sigma_1 \ \mathbf{refuses} \ \{\lambda\} \\ & \implies \exists p \in (s \ \mathbf{after} \ \sigma_1), p \stackrel{\lambda}{\not\Rightarrow} \\ & \implies \chi \in (DC(s) \ \mathbf{after} \ \sigma_1{\cdot}\lambda) \ \ \text{(for the definition of DC(s))} \\ & \implies \forall \sigma' \in L_\delta^*, \sigma_1{\cdot}\lambda \preceq \sigma' : out(DC(s) \ \mathbf{after} \ \sigma') = L_U \cup \{\delta\} \end{aligned}$$

In particular for $\sigma' = \sigma$:

$$out(DC(s) \ \mathbf{after} \ \sigma) \not\subseteq out(s' \ \mathbf{after} \ \sigma) \implies \neg(DC(s) \ \mathbf{ioco}_{Rtraces(s)} \ s')$$

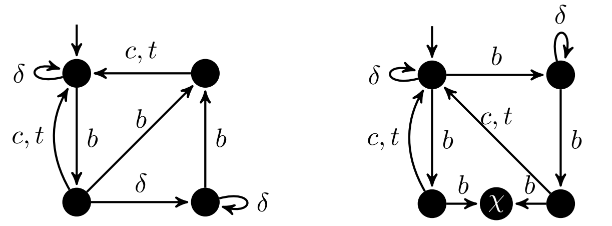
and thus $DC(s) \notin SUT(s')$ as required.

□

Theorems 4.5 and 4.6 give a way to relate specifications; if they are in such a relation then the implementations that are conforming to one are also conforming to the other. This is a starting point for a test selection method, which, given a specification s aims at finding another specification s' that is in such a relation with s. Thus deriving tests from s gives sound tests for s'. (see section 6). Furthermore, it is possible to use **wioco** as implementation relation, i.e. testing the conformance of an non-input enabled implementation w.r.t. a specification. This approach leads to a relation similar to alternating simulations [1].

5 Required Traces Automata

A *suspension automaton* [19], or deterministic *quiescent labelled transition system* [14], is a model built from a specification that represents exactly the set of traces used to test for **ioco** conformance. It is obtained by adding δ-loops for all quiescent states and then determinizing the obtained model. This approach does not work for **uioco** or $\mathbf{ioco}_{Rtraces(s)}$ because the information on underspecified traces is lost during the determinization process.



(a) Suspension automaton of s' in figure 2b.

(b) Demonic completion of s'.

Figure 9: Suspension automata are not suitable to describe u-traces or r-traces.

Example 5.1. *Figure 9a shows the determinization of s' (figure 2b). In s' the trace $b{\cdot}b$ is underspecified because of the left branch, thus, in a **uioco** conforming implementation, any behavior is allowed after it. This information is lost after the determinization.*

In order to obtain a similar result for r-traces, we can first make explicit the underspecifications, e.g. using demonic completion (figure 9b), and then apply the same steps used to construct the suspension automaton, i.e. adding δ-transitions, if necessary, and determinizing. The obtained model is a deterministic quiescent labelled transition system, with the nice property that all the s-traces that are not u-traces lead to a state that is trace equivalent to χ.

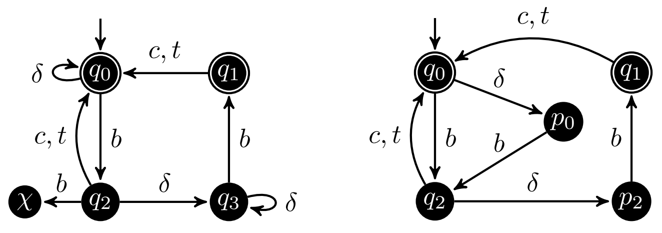
Now we can mark all the states as *accepting*, except for those that enable $L_U \cup \{\delta\}$ and those that are reached only by δ transitions. We need to mark some states as accepting because the set of r-traces is not prefix closed. Note that states that are trace equivalent to χ are not marked this way.

As a next step we can remove from the model those states that are trace equivalent with the chaos state χ. Once one of them is reached it is not possible to move to an accepting state, thus the information added by these states is not useful. At this point the language accepted by the automaton describes a *superset* of *Rtraces*(s), because we still need to handle the special cases in which a state with a δ-loop could be marked as accepting.

In figure 10a it is explicit why we need to manipulate the automaton obtained so far: the initial state, for example, is marked as accepting, in fact ϵ is an r-trace. Unfortunately this will also add the trace δ to the accepted language, because of the δ-loop. Furthermore, as described in section 4, sequences of δ

must be reduced to a single δ. In order to consider these two cases we apply the following steps to the required traces automaton: i) replace δ-transitions such as $q \xrightarrow{\delta} q_1$ where $q \neq q_1$ with $q \xrightarrow{\delta} p$ and $p \xrightarrow{\lambda} q_2$ where p is a new state, $q_2 \in Q$ and $q_1 \overset{\lambda}{\Rightarrow} q_2$ with $\lambda \neq \delta$; ii) replace δ-loop transitions such as $q \xrightarrow{\delta} q$ with $q \xrightarrow{\delta} p$ and $p \xrightarrow{\lambda} q_1$ where p is a new state and $q \overset{\lambda}{\Rightarrow} q_1$ with $\lambda \neq \delta$.

Example 5.2. *Figure 10 shows the determinization of the demonic completion of figure 9b and the required-traces automaton of s'. Figure 10b gives the*



(a) A determinization of the demonic completion of s'.

(b) Required-trace automaton of s'

Figure 10: Determinization of the demonic completion and required-traces automaton of s'. A double circle represents an accepting state.

automaton obtained by removing from the model of figure 10a those states that behave like the chaotic state χ, i.e. those states that, ones they have been reached, allow any behavior, and then applying step i) and step ii) described in this section. First step i) is applied to $q_2 \xrightarrow{\delta} q_3$ adding a new state p_2. The state q_3 is then removed because it is not reachable from the initial one. Step ii) is then applied to q_0 creating a new state p_0.

New states added this way are not accepting states, because the only transitions that are reaching them are δ-transitions. After all δ-loops have been removed, all δ-transitions reach new non-accepting states and no δ-transitions leave those states. Thus no traces with sequences of δ and no traces ending in δ are accepted by the language of the final automaton. We call the final result the *required-traces automaton* and indicate it as $RTA(s)$. All added transitions are justified by existing traces, i.e. $q \xrightarrow{\delta} p \xrightarrow{\lambda} q_2$ with $\lambda \neq \delta$ is added only if $q \overset{\delta \cdot \lambda}{\Longrightarrow} q_2$. Thus all the added traces of the final automaton are accepted also by the initial one.

Theorem 5.1. *Let $s \in \mathcal{LTS}(L_I, L_U)$ and $\sigma \in L_\delta^*$, then $\sigma \in Rtraces(s) \iff \sigma$ is accepted by $RTA(s)$.*

Proof. $(\Rightarrow)$

$$\sigma \in Rtraces(s) \implies \sigma \in Straces(DC(s)) \qquad \text{(proposition 3.2.3)}$$

thus $DC(s)$ **after** σ is not empty. Determinization preserves traces, so, let $DC^d(s)$ be the determination of the demonic completion of s, there exists a

unique state $q \in (DC^d(s)$ **after** $\sigma)$.
q is marked as accepting because of definition 4.1 and q is not removed from the automaton because, being $out(DC^d(s)$ **after** $\sigma) \neq L_U \cup \{\delta\}$, q is not trace equivalent to χ.

Furthermore σ does not contain sequences of δ and does not end with δ, thus, after δ-sequences removal, the state that is reached after σ remains the same: $q \in (RTA(s)$ **after** $\sigma)$ that is still an accepting state.

$(\Leftarrow)$

If σ is accepted by $RTA(s)$ then $q \in (RTA(s)$ **after** $\sigma)$ is an accepting state and thus:

1. σ does not contain sequences of δ
2. σ does not end with δ
3. $out(q) \neq L_U \cup \{\delta\}$
4. $\sigma \in Utraces(s)$

Points (1) and (2) are supported by the δ-sequences removal step. Point (3) is supported by the fact that q is marked as accepting and, being $RTA(s)$ deterministic, (3) $\implies$ $out(RTA(s)$ **after** $\sigma) \neq L_U \cup \{\delta\}$. Point (4) is given by the fact that the required traces automaton construction does not add any trace to $Straces(s)$ (those that are added during demonic completion are removed later while removing states that are trace equivalent to χ), thus $\sigma \in Straces(s)$. Furthermore s-traces that are not u-traces are removed from the specification while removing states that are trace equivalent to χ. Thus $\sigma \in Utraces(s)$.

$$(1), (2), (3) \text{ and } (4) \implies \sigma \in Rtraces(s)$$

□

5.1 The Algorithm

The required traces automaton of a specification s, can be constructed using algorithm 1. The algorithm applies the steps described in the previous sections: lines 1 and 2 build the demonic completion (with δ transitions) of the initial specification s, making explicit the underspecification implied by **uioco** relation. Line 3 determinizes the obtained input-output transition system: determinization preserves traces, thus no traces are lost in the process. Line 5 marks the states which do not enable all output actions as accepting. The language accepted by the obtained automaton is not yet the set of r-traces due to possible δ-sequences. For this reason we first remove states that are trace equivalent to χ at lines 7 and 8 and then we remove δ-sequences from line 10 to 14.

Algorithm 1 Find the required traces automaton of $s \in \mathcal{LTS}(L_I, L_U)$

```
1: add loops $s \xrightarrow{\delta} s$ for all quiescent states
2: build $DC(s)$, the demonic completion of $s$
3: determinize $DC(s)$ obtaining a new input-output transition system $DC^d(s) = \langle Q, L_I, L_U, T, q_0 \rangle$
4: for each $q \in Q$ do
5:     if $(out(q) \neq L_U \cup \delta) \wedge (\exists p \mid p \xrightarrow{\lambda} q \wedge \lambda \neq \delta \wedge \lambda \neq \tau)$ then mark $q$ as accepting
6: for each $q \in Q$ do
7:     if $q$ is trace equivalent to chaos state $\chi$ then mark $q$ as chaotic
8: remove all chaotic states from $Q$
9: for each $(q, \delta, q_1) \in T$ do
10:    add a new state $p$ to $Q$                    {δ sequences optimization steps 1 and 2}
11:    add $(q, \delta, p)$ to $T$
12:    for each $a \in L_I$ such that $q_1 \overset{a}{\Rightarrow} q_2$ do
13:        add $(p, a, q_2)$ to $T$                  {if s is a valid specification then $out(q) \not\subseteq L_U$}
14:    remove $(q, \delta, q_1)$ from $T$
```

6 Test Selection for r-traces

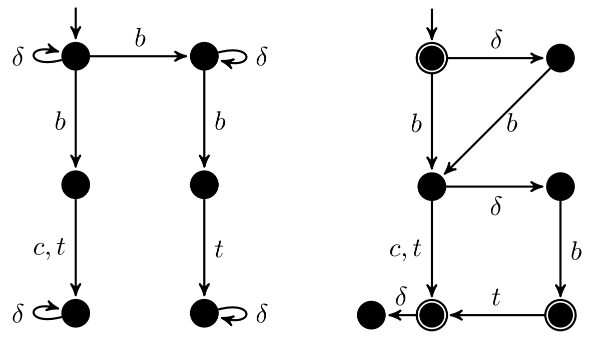
The set of r-traces is still usually an infinite set, even after the removal of δ-sequences. In checking the conformance of an implementation with a given specification we can only use some test cases, i.e. constructed from some r-traces, because of the limited time we have.

Considering figure 1, we can find a possible approach to treat this issue. Given a specification s, by selecting a proper finite subset of the r-traces set as a test suite, we test the conformance of an implementation to another specification s' which allows more conforming implementations than s. Being this test suite finite, we are able to exhaustively test for **uioco** conformance between the implementation and s'.

In other words, given a specification s and a finite set R subset of $Rtraces(s)$, we define s_R as a labelled transition system which has R as set of r-traces and in which the output behavior after observing a trace in R covers the output behavior after observing that trace in s. Formally:

Definition 6.1. *Given $s \in \mathcal{LTS}(L_I, L_U)$ and a finite set $R \subseteq Rtraces(s)$, then $s_R \in \mathcal{LTS}(L_I, L_U)$ is a labelled transition system such that $Rtraces(s_R) = R$ and $\forall \sigma \in R : out(s$ **after** $\sigma) \subseteq out(s_R$ **after** $\sigma)$.*

Example 6.1. *Let us, for instance, consider specification s' of figure 2b. Its r-traces set is infinite (see its required-traces automaton, figure 10b). We select $R = \{\epsilon, b{\cdot}c, b{\cdot}t, \delta{\cdot}b{\cdot}c,$
$\delta{\cdot}b{\cdot}t, b{\cdot}\delta{\cdot}b, \delta{\cdot}b{\cdot}\delta{\cdot}b, b{\cdot}\delta{\cdot}b{\cdot}t, \delta{\cdot}b{\cdot}\delta{\cdot}b{\cdot}t\}$. A specification which follows definition 6.1, given s' and R is the one of figure 11a. The required trace automaton of s_R is the one of figure 11b. It is a directed acyclic graph. An implementation can be exhaustively tested with respect to s_R. Such a specification can also be found by first unfolding the cycles in s' and then checking that the r-traces set of the new*



(a) Specification s_R. (b) Required-traces automaton of s_R.

Figure 11: A specification s_R whose set of r-traces is a finite subset of $Rtraces(s')$. It is weaker than s'.

specification is contained in $Rtraces(s')$.

A specification like the one defined in definition 6.1 exists (theorem 6.1) and s is **wioco** conforming to it (theorem 6.2).

Theorem 6.1. *Given* $s \in \mathcal{LTS}(L_I, L_U)$ *and a finite set* $R \subseteq Rtraces(s)$*, then there exists a specification* s_R *as defined in definition 6.1.*

Proof. s_R can be constructed starting from s and R:

1. construct the deterministic tree automaton A which accepts R;
2. $\forall \sigma \in R, \forall \lambda \in out(s \textbf{ after } \sigma)$ such that $\lambda \notin out(A \textbf{ after } \sigma)$ add $q \xrightarrow{\lambda} \chi$ where $q \in (A \textbf{ after } \sigma)$ and χ is a chaotic state;
3. add δ-loops to quiescent states;
4. transform all accepting states into normal states.

From the construction it is trivial that:

$$\forall \sigma \in R : out(s \textbf{ after } \sigma) = out(s_R \textbf{ after } \sigma)$$

Furthermore A is deterministic and accepts R which is a set of r-traces, thus

$$\begin{aligned} \forall \sigma \in R : \sigma \in Utraces(s_R) &= Straces(s_R) \\ &\wedge \sigma \text{ does not end with } \delta \\ &\wedge out(s_R \textbf{ after } \sigma) \neq L_U \cup \{\delta\} \end{aligned}$$

It is possible to note, from the construction of s_R, that traces which are not in R are underspecified or they lead to a chaotic state. Thus they cannot be r-traces. For definition 4.1, R is the set of r-traces of s_R.

□

Theorem 6.2. *Given $s \in \mathcal{LTS}(L_I, L_U)$, a finite set $R \subseteq Rtraces(s)$ and s_R as defined in definition 6.1, then s **wioco** s_R.*

Proof. We prove that, $\forall \sigma \in Rtraces(s_R)$:

1. $out(s\ \textbf{after}\ \sigma) \subseteq out(s_R\ \textbf{after}\ \sigma)$
2. $\forall \sigma_1 \preceq \sigma : Rin(s_R, \sigma_1) \subseteq in(s\ \textbf{after}\ \sigma_1)$

1. is trivial from the construction of s_R.
Let σ_1 be a prefix of an r-trace $\sigma \in Rtraces(s_R)$:

$$\begin{aligned}
\lambda \in Rin(s_R, \sigma_1) &\Longrightarrow s_R\ \textbf{after}\ \sigma_1\ \textbf{must}\ \{\lambda\} \wedge \exists \sigma' \in Rtraces(s_R) : \sigma_1 \cdot \lambda \preceq \sigma' \\
&\Longrightarrow \sigma' \in Rtraces(s) \\
&\Longrightarrow \sigma' \in Utraces(s) \\
&\Longrightarrow \sigma_1 \cdot \lambda \in Utraces(s) \qquad \text{(for lemma 2.2.1)} \\
&\Longrightarrow s\ \textbf{after}\ \sigma_1\ \textbf{must}\ \{\lambda\}
\end{aligned}$$

Thus $Rin(s_R, \sigma_1) \subseteq in(s\ \textbf{after}\ \sigma_1)$ as required. □

There are at least two methods to get an s_R from a given specification s. It can be obtained by selecting a finite subset of $Rtraces(s)$ or by modifying the specification s by applying some transformations that do not remove any implementation from $SUT(s)$, i.e. soundness is preserved.

Following the first method, it is possible to provide heuristics that can be used to select traces among r-traces in order to reach a certain level of coverage. In this way s and s_R can be quantitatively compared, giving a value to the difference between the initial specification and the tested, weaker one. For example trace distance [5] can be applied in this sense.

Acting directly on the specification s by modifying it can be done with the use of three transformations: remove an input transition, adding an output transition and unfolding a cycle (to a certain depth). The final aim of applying these transformations is finding a specification s_R that is graphically represented by a finite directed acyclic graph. In this situation the set of r-traces is finite and so it is possible to exhaustively test implementations w.r.t. it.

Explaining in details these two methods is not in the purpose of this paper, but it is an interesting extension of this theory that can be investigated in future work.

7 Conclusions

In this work we have addressed two problems that arise while dealing with test selection for the **ioco** framework: the interdependence of test suites currently used for testing the conformance of an implementation with a given specification, i.e. s-traces and u-traces, and the lack of relations on specifications that can be used as a test selection method, i.e. relations on non-input enabled specifications.

We introduced a new set of traces, r-traces, which tighten the set of u-traces in order to make its elements independent from each other without loosing testing power. We used this new set of traces to define a new relation between specifications that can be used as implementation relation for testing non-input enabled implementations. We gave a way to obtain the r-traces set from a given specification using a canonical representation in the form of an automaton and we gave an algorithm for constructing this automaton.

As future work, a study on quantitative aspects related to the qualitative outcomes of this paper should be considered. By giving a distance function either on specifications or on r-traces sets, it will be possible to compute a coverage on them, i.e. estimating the testing power of a selected subset of a given r-traces set or of a specification which is in **wioco** relation with a given one. In order to define such functions, the way to obtain these sets or these specifications should be first deeply analyzed, exploiting the methods that have been introduced in section 6.

References

[1] R. Alur, T. Henzinger, O. Kupferman, and M. Vardi. Alternating refinement relations. In D. Sangiorgi and R. de Simone, editors, *CONCUR'98 Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 163–178. Springer Berlin / Heidelberg, 1998.

[2] Axini. http://www.axini.com.

[3] A. Belinfante. JTorX: A Tool for On-Line Model-Driven Test Derivation and Execution. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[4] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A Symbolic Test Generation Tool. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2002*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer, 2002.

[5] L. M. G. Feijs, N. Goga, S. Mauw, and J. Tretmans. Test Selection, Trace Distance and Heuristics. In *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, TestCom '02, pages 267–282, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[6] L. Frantzen and J. Tretmans. Model-Based Testing of Environmental Conformance of Components. In F. S. de Boer, M. M. Bosangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects – 5^{th} Int.\ Symposium FMCO 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2007.

[7] L. Frantzen, J. Tretmans, and T. A. C. Willemse. Test generation based on symbolic specifications. In *FATES 2004, number 3395 in LNCS*, pages 1–15. Springer-Verlag, 2005.

[8] N. Goga and F. Moldoveanu. Test selections and coverages. In *17th Annual IEEE Canadian Conference on Electrical and Computer Engineering*, pages 707–710. IEEE Canada, 2004.

[9] A. Hartman and K. Nagin. The AGEDIS Tools for Model Based Testing. In *Int.\ Symposium on Software Testing and Analysis – ISSTA 2004*, pages 129–132. ACM Press, 2004.

[10] J. He and K. J. Turner. Protocol-Inspired Hardware Testing. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, *Int.\ Workshop on Testing of Communicating Systems 12*, pages 131–147. Kluwer Academic Publishers, 1999.

[11] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing Real-Time Systems Using UPPAAL. In R. M. Hierons, J. P. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer-Verlag, 2008.

[12] C. Jard and T. Jéron. TGV: Theory, Principles and Algorithms: A Tool for the Automatic Synthesis of Conformance Test Cases for Non-Deterministic Reactive Systems. *Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[13] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *In 11th International SPIN Workshop on Model Checking of Software (SPIN04), volume 2989 of LNCS*, pages 109–126. Springer, 2004.

[14] M. Timmer, H. Brinksma, and M. I. A. Stoelinga. Model-Based Testing. In M. Broy, C. Leuxner, and C. A. R. Hoare, editors, *Software and Systems Safety: Specification and Verification*, volume 30 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 1–32. IOS Press, Amsterdam, Apr. 2011.

[15] J. Tretmans. Model-Based Testing and Some Steps towards Test-Based Modelling. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems*, volume 6659 of *Lecture Notes in Computer Science*, pages 297–326. Springer Berlin / Heidelberg, 2011.

[16] J. Tretmans and E. Brinksma. TorX: Automated Model-Based Testing. In A. Hartman and K. Dussa-Zieger, editors, *First European Conference on Model-Driven Software Engineering*, pages 31—-43. Imbuss, M{ö}hrendorf, Germany, Dec. 2003.

[17] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional Testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software*

Testing – FATES 2003, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, 2004.

[18] M. van der Bijl, A. Rensink, and J. Tretmans. Action refinement in conformance testing. In *Proceedings of the 17th IFIP TC6/WG 6.1 international conference on Testing of Communicating Systems*, TestCom'05, pages 81–96, Berlin, Heidelberg, 2005. Springer-Verlag.

[19] T. Willemse. Heuristics for ioco-Based Test-Based Modelling. In L. Brim, B. Haverkort, M. Leucker, and J. van de Pol, editors, *Formal Methods: Applications and Technology*, volume 4346 of *Lecture Notes in Computer Science*, pages 132–147. Springer Berlin / Heidelberg, 2007.