Test Selection for the ioco Framework

Michele Volpato and Jan Tretmans
Institute for Computing and Information Sciences
Radboud University Nijmegen
Nijmegen, The Netherlands
Email: {m.volpato, tretmans}@cs.ru.nl

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

Keywords—model-based testing; ioco theory; test selection;

I. 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 [1]. 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 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 [2].

In this paper we will address the test selection problem for the ioco framework [3]. 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 example see [4] [5] [6] [7], and it has been implemented in several model-based testing tools such as TorX [8], JTorX [9], TGV [10], STG [11], TestGen [12], Uppaal-Tron [13], the Agedis Tool set [14], and the commercial tool Axini Test Manager [15].

In the original ioco theory, test cases are represented by 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 [16] 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 satisfies some compositionality properties of labelled transition systems.

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 two different u-traces could give the same result on every possible implementation.

Within the ioco framework, another way to select test cases from a specification is testing the conformance of an implementation with a different, weaker specification [17]. 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 III).

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, which elements are independent, ii) a variation of ioco, based on r-traces, which can be used for selecting specifications for the test
selection problem and iii) the formulation of a canonical representation of r-traces and an algorithm to obtain it from a specification.

Testing conformance on r-traces is equivalent to test it 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. The same set can be used also to qualitatively compare specifications through a variation of ioco, called wioco, that allows non input-enabled implementations. This comparison permits to relate specifications in a preorder according to their set of conforming implementations, giving the possibility to reason about one specification being stronger than another one for the test selection problem.

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. We will also mention two methods that can be used to reduce the set of possible tests. One method deals directly with test cases, i.e. r-traces, the other, instead, aims at modifying the initial specification in order to find another one which is in wioco relation with it.

II. 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 give some well-know definitions regarding labelled transition systems and ioco testing theory introduced in [3].

A labelled transition system (with inputs and outputs) is a 5-tuple \((Q, L, \delta, \lambda, q_0)\), where \(Q\) is a set of states, \(L\) and \(\lambda\) are the set of input and output labels or actions, respectively, such that \(L \cap \lambda = \emptyset\), \(T \subseteq Q \times (L \cup \lambda \cup \{\tau\}) \times Q\) is the transition relation and \(q_0\) is the initial state. The special label \(\tau\) represents an internal unobservable transition.

We use the name of the labelled transition system or its initial state interchangeably. We write \(\delta \rightarrow q'\) rather then \((q, \lambda, q') \in T\) and \(\exists q' \in Q : q \xrightarrow{\lambda} q'\) is abbreviated to \(q \xrightarrow{\lambda}\).

We use \(L\) for \(L \cup \tau\). We denote the class of all labelled transition systems over \(L\) and \(\tau\) as LTS(L_1, L_2).

Let \(q, q'\) be states of a labelled transition system \(s \in LTS(L_1, L_2), \sigma \in L^*, \lambda \in L \) and \(\epsilon\) be the empty sequence, then we define:

\[
\begin{align*}
q \xrightarrow{\delta} q' & \iff q = q' \text{ or } q \xrightarrow{\tau} \ldots \xrightarrow{\tau} q' \\
q \xrightarrow{\lambda} q' & \iff \exists q_1, q_2 : q \xrightarrow{\lambda} q_1 \xrightarrow{\lambda} q_2 \xrightarrow{\lambda} q' \\
q \xrightarrow{\sigma} q' & \iff q \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} q'
\end{align*}
\]

where \(\sigma = \lambda_1 \ldots \lambda_n\).

The set of traces of \(s\) is defined as \(\text{traces}(s) = \{\sigma \in L^* | s \xrightarrow{\sigma}\}\) where \(s \xrightarrow{\sigma}\) abbreviates \(3q' \in Q : s \xrightarrow{\sigma} q'\).

We denote the set of states that are reachable from a given state \(q\) via a \(\sigma\) as \(q\ \text{after}\ \sigma = \{q' | q \xrightarrow{\sigma} q'\}\) and by extension, given a set of states \(P\) as \(P\ \text{after}\ \sigma = \bigcup\{q \text{ after } \sigma | q \in P\}\).

A set of actions \(A\) is refused by a set of states \(P\) if \(\exists q' \in P, \forall q \in A \cup \{\tau\} : q \xrightarrow{\tau} q'\) and we write it as \(P\ \text{refuses}\ A\). A state \(q\) is quiescent if \(\{q\}\) refuses \(L\).

Let \(\delta \notin L_1 \cup L_2\) and \(s_\delta\) be the labelled transition system to which we add a \(\delta\)-loop transition \(q \xrightarrow{\delta}\ q\) to all quiescent states, then the set of s-traces is

\[\text{Straces}(s) = \{\sigma \in L_\delta^* | s_\delta \xrightarrow{\sigma}\}\]

where \(L_\delta^*\) is the set of sequences of actions in \(L \cup \delta\). From now on we will usually include \(\delta\)-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 \(\text{out}(P) = \{\lambda \in L \cup \{\delta\} | \exists q' \in P : q' \xrightarrow{\lambda}\}\).

If \(\forall q \in Q, \lambda \in L_1 : q \xrightarrow{\lambda}\) we say that the system is input-enabled. The class of all input-enabled labelled transition systems over \(L_1\) and \(L_2\) is denoted \(\text{IOTS}(L_1, L_2)\).

Finally, given a set of traces \(F\), an implementation \(i \in \text{IOTS}(L_1, L_2)\) and a specification \(s \in \text{LTS}(L_1, L_2)\) we define the implementation relations as:

\[
i \xrightarrow{\text{ioco}_F} s \iff \forall \sigma \in F : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)\]

\[
i \xrightarrow{\text{ioco}} s \iff i \xrightarrow{\text{ioco}_{\text{Straces}(s)}} s\]

\[
i \xrightarrow{\text{tor}} s \iff i \xrightarrow{\text{ioco}_{L_\delta^*}} s\]

In [16] a subset of \(\text{Straces}(s)\), called \(\text{Utraces}(s)\) is introduced and studied. It differs from \(\text{Straces}(s)\) on so called underspecified traces:

\[
\text{Utraces}(s) = \{\sigma \in \text{Straces}(s) | \forall \sigma_1, \sigma_2 \in L_\delta^*, \lambda \in L_1 : \sigma = \sigma_1 \cdot \lambda \cdot \sigma_2 \text{ implies not } (s \text{ after } \sigma_1 \text{ refuses } \{\lambda\})\}
\]

We write \(i \xrightarrow{\text{ioco}} s\) for \(i \xrightarrow{\text{ioco}_{\text{Utraces}(s)}} s\).

The relations ioco and uioco are defined only on input-enabled implementations, but in this paper we consider also relations among specifications (i.e. labelled transition systems) Thus we need to consider also a variation of uioco that handles non input-enabled implementations. The work in [4] inspires a possible solution, also similar to alternating simulations [18], to the non input-enabled specification issue by using the set of inputs after a trace \(\sigma\), in(\(p\) after \(\sigma\), i.e. 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 uioco conforming),
but it also must allow any input that is not underspecified by the specification after each u-trace. Formally: let \( s, s' \in LTS(L_I, L_U) \), \( \sigma \in L_\delta \), \( P \) be a set of states and \( \lambda \in L \). Then

1. \( P \) must \( \lambda =_{def} \forall p \in P : p \lambda \)
2. \( in(P) =_{def} \{ a \in L_I \mid P \text{ must } a \} \)
3. (uioco between specifications)
   \[
   s \text{ uioco } s' =_{def} \forall \sigma \in Utraces(s'): \\
   a) \text{ out}(s \text{ after } \sigma) \subseteq \text{ out}(s' \text{ after } \sigma) \land \\
   b) \text{ in}(s \text{ after } \sigma) \supseteq \text{ in}(s' \text{ after } \sigma)
   \]

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

III. 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 in 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. Being able to restrict the set of test cases to those that allow to find failures with a certain (possibly high) accuracy, is the goal of test selection.

Many selection techniques have been studied in the domain of conformance testing of labelled transition systems [19], such as heuristic criteria selection [20].

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. If, on the contrary, we test \( i \) only for a smaller and finite number of traces, which correspond to a specification \( s' \) that is weaker than \( s \), then we can say that \( i \) is conforming to \( s \) with a certain probability 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

\[
SUT(s) \equiv \{ i \in IOTS(L_I, L_U) \mid i \text{ uioco } s \}
\]

Basically, a specification \( s \) refines a specification \( s' \) if \( SUT(s) \subseteq SUT(s') \); this weakness relation leads to the definition of a preorder among specifications. A good choice for \( s' \) would be a specification for which the area \( SUT(s') \setminus SUT(s) \) is small. 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.

This paper focuses on the qualitative aspect, thus the goal is to find, or define, a relation \( \mathcal{R} \), defined on labelled transition systems, such that \( s \mathcal{R} s' \iff SUT(s) \subseteq SUT(s') \).

The trivial adaptation of uioco for not input-enabled implementations does not satisfy this constraint: 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 III.1), the other way around is not always valid. Example III.1 shows it.

**Theorem III.1.** Let \( s, s' \in LTS(L_I, L_U) \) be two specifications over the same input and output sets,

\[
s \text{ uioco } s' \implies SUT(s) \subseteq SUT(s')
\]

**Example III.1.** Figure 2 shows a situation in which \( SUT(s) \subseteq SUT(s') \) and \( s \text{ uioco } s' \), figure 2a represents a specification on \( LTS(\{a, b\}, \{x, y\}) \) with an underspecified trace \((axb)\). This underspecified trace has been made explicit in the specification \( s' \), described by figure 2b. Having made only explicit the underspecification implies \( SUT(s) = SUT(s') \), but \( s \text{ uioco } s' \), see definition in section II, i.e.

\[
\forall \sigma = ax, \emptyset = in(s \text{ after } \sigma) = in(s' \text{ after } \sigma) = \{b\}.
\]

In conclusion, given a specification \( s \) with some underspecified traces, making explicit the underspecified traces (for example by applying demonic completion, that will be better explained in section III-A) will provide a new input-enabled specification \( s' \). Implementations that are uioco w.r.t. \( s \) are also uioco w.r.t. \( s' \), but \( s \text{ uioco } s' \).
A. Demonic Completion

Example III.1 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 [16] goes well for this purpose. It introduces a chaotic state, a state that accepts any input and may generate any output.

Definition III.1 (Chaotic state \( \chi \)). A chaotic state \( \chi \) is a state with the following properties:

\[
\begin{align*}
&\forall \lambda \in L_I \cup L_U : \chi \xleftarrow{\lambda} \chi \\
&\forall a \in L_I : \chi \xrightarrow{a,\chi} \chi
\end{align*}
\]

where \( L_I \) and \( L_U \) are the sets of input and output labels respectively and \( \delta \) represents quiescence.

A chaotic state can be defined in many ways, figure 3 shows a nondeterministic and a deterministic one.

![Figure 3: Graphical representations of chaotic behavior.](image)

The demonic completion of a specification is the input-output transition system obtained by adding a chaotic state \( \chi \) to that specification such that all non-specified inputs lead to \( \chi \). In the following definition the deterministic version of a chaotic state (figure 3b) is used.

Definition III.2. Given a specification \( s = (Q, L_I, L_U, T, q_0) \), the demonic completion \( DC(s) \) of \( s \) is a tuple \( (Q', L_I, L_U, T', q_0) \) where:

\[
\begin{align*}
Q' &= Q \cup \{\chi, \chi_1\} \text{ with } \chi, \chi_1 \not\in Q \\
T' &= T \cup \{(q, a, \chi) \mid q \in Q, a \in L_I, q \not\xrightarrow{a}, q \not\xrightarrow{a}\} \\
&\cup \{(\chi, \delta, \chi_1)\} \\
&\cup \{(\chi_1, a, \chi) \mid a \in L_I\} \\
&\cup \{(\chi, \lambda, \chi) \mid \lambda \in L_I \cup L_U\}
\end{align*}
\]

The demonic completion of a labelled transition system is an input-output transition system, i.e. it is input-enabled. Moreover once the state \( \chi \) 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 III.2.** Let \( s, s' \in LTS(L_I, L_U) \) be two specifications, and \( DC(s), DC(s') \) be their demonic completions,

1) \( DC(s), DC(s') \in TOTS(L_I, L_U) \) and so, from [3]:
   a) \( DC(s) \) uioco \( DC(s') \) \( \iff \) \( DC(s) \) ioco \( DC(s') \)
   b) \( DC(s) \) uioco \( DC(s') \) \( \iff \) \( Straces(DC(s)) \subseteq Straces(DC(s')) \)

2) \( DC(s) \) uioco \( s \)

3) \( \sigma \in Straces(s) \implies \sigma \in Straces(DC(s)) \)

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 \( \sigma \) is underspecified only if all the states reachable after \( \sigma \) do not enable that input. We can see an example in figure 4.

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

\[
\{x, \delta\} = \text{out}(DC(s) \text{ after } \sigma) \not\subseteq \text{out}(s \text{ after } \sigma) = \{x\}
\]

In fact, \( DC(s) \) after \( a \cdot \delta \) contains the chaotic state \( \chi \) 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 III.3). Thus \( \leq_{ior} \) on demonically completed specifications is a valid relation that can be used for our purpose.

**Theorem III.3.** Let \( s, s' \in LTS(L_I, L_U) \) be two specifications over the same input and output sets,

\[
SUT(s) \subseteq SUT(s') \iff DC(s) \leq_{ior} DC(s')
\]
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.

IV. THE SET OF REQUIRED TRACES

In the previous section we found 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 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.

In this section we present the set of required traces. Required traces are, basically, the only traces that are needed of Utraces because of an output after quiescence. Required traces are, basically, the only traces that are needed of Utraces such as ioco conforming implementations.

The set of required traces of a specification \( s \) is a subset of Utraces(\( s \)). First we define a subset of Utraces(\( s \)) as intermediate step (definitions IV.2 and IV.3) and then we refine it obtaining the set of required traces (definition IV.4).

The idea behind following definitions is that we want to remove from the set of u-traces 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.

Given a specification \( s \in LTS(L_I, L_U) \), we say that a trace \( \sigma \in Utraces(\( s \)) \) is uioco-specifying if and only if it is possible to build an input-output transition system \( i \in IOTS(L_I, L_U) \) such that \( i \) uioco \( s \) because of an output after \( \sigma \). Formally:

**Definition IV.1** (uioco-specifying trace). Let \( s \in LTS(L_I, L_U) \). We say that a trace \( \sigma \in Utraces(\( s \)) \) is uioco-specifying in \( s \) if and only if

\[
\exists i \in IOTS(L_I, L_U) : \text{out}(i \ \text{after} \ \sigma) \not\subseteq \text{out}(s \ \text{after} \ \sigma)
\]

The set of uioco-specifying traces of \( s \) is the subset of Utraces(\( s \)) that contains all and only the traces that are uioco-specifying in \( s \).

**Definition IV.2.** Let \( s \in LTS(L_I, L_U) \).

\[ UiocoSpec(\( s \)) =_{def} \{ \sigma \in Utraces(\( s \)) \mid \sigma \text{ is uioco-specifying in } \sigma \} \]

An equivalent definition of UiocoSpec(\( s \)), that does not depend on implementations is:

**Definition IV.3.** Let \( s \in LTS(L_I, L_U) \).

\[ UiocoSpec(\( s \)) =_{def} \{ \sigma \in Utraces(\( s \)) \mid \sigma \text{ does not end with } \delta \land out(s \text{ after } \sigma) \neq L_U \cup \{\delta\} \}
\]

Those two definitions are equivalent (lemmas IV.1), thus we will use one definition or the other without distinction.

**Lemma IV.1.** \( UiocoSpec(\( s \)) = UiocoSpec'(\( s \)) \).

There is still an issue that must be considered about the set UiocoSpec(\( s \)). With the current definition of Utraces(\( s \)), elements of UiocoSpec(\( s \)) are not independent from each other, i.e. there could be two elements which spot the same wrong implementations. This is mainly due to repetitive quiescence.

A. \( \delta \) sequences optimization

In [21] and [22] repetitive quiescence has been addressed with the result that, in a labelled transition system, after adding \( \delta \)-loops to quiescent states, the behaviour after observing quiescence is the same if quiescence is observed again. Thus 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 \( \delta \) transitions, because the result for those u-traces will be the same as the u-trace obtained by collapsing the subsequences of consecutive \( \delta \) transitions to a single one.

Thus we can remove from the set UiocoSpec(\( s \)) those traces that contain sequences of quiescence. This optimization leads to the definition of Rtraces(\( s \)):

**Definition IV.4.** Let \( s \in LTS(L_I, L_U) \).

\[ Rtraces(\( s \)) =_{def} \{ \sigma \in UiocoSpec(\( s \)) \mid \delta \cdot \delta \text{ is not a substring of } \sigma \} \]

We say that an element belonging to the set of Rtraces(\( s \)) of an arbitrary labelled transition system \( s \) is an r-trace.

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. That means the we removed too many traces. Theorem IV.2 proves that this is not the case for Rtraces(\( s \)).

**Theorem IV.2.** Let \( s \in LTS(L_I, L_U) \) be a specification and \( i \in IOTS(L_I, L_U) \) be an input-output transition system.

\[ i \text{ uioco } \iff i \text{ ioco } \text{Rtraces}(\( s \)) \text{ s} \]

After the optimization, it is not possible to find a set \( A \subset Rtraces(\( s \)) \) for which it is valid that \( i \text{ uioco } \iff i \text{ ioco } A \text{ s} \), i.e. all \( \sigma \in Rtraces(\( s \)) \) are necessary for uioco checking. Another way to explain this concept is that for two different r-traces \( \sigma \) and \( \sigma' \) there exists an implementation \( i \) such that \( i \) fails uioco after \( \sigma \) but not after \( \sigma' \) and vice versa. Thus r-traces are independent of one another. Formally:

**Theorem IV.3.** Let \( s \in LTS(L_I, L_U) \) be a specification, \( \forall \sigma, \sigma' \in Rtraces(\( s \)) \text{ such that } \sigma \neq \sigma' \).

\[ \exists i \in IOTS(L_I, L_U) : \]

\[ \text{out}(i \ \text{after} \ \sigma) \not\subseteq \text{out}(s \ \text{after} \ \sigma) \]

\[ \land \text{out}(i \ \text{after} \ \sigma') \subseteq \text{out}(s \ \text{after} \ \sigma') \]
Theorem IV.3 is not valid for traditional s-traces or u-traces in general. For example if we consider the specification \( s \) of figure 4a, 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 \( \iota \) such that
\[
\text{out}(i \text{ after } \delta \cdot a \cdot x) \not\subseteq \text{out}(i \text{ after } \delta \cdot a \cdot x) \land \\
\text{out}(i \text{ after } \delta \cdot \delta \cdot a \cdot x) \subseteq \text{out}(i \text{ after } \delta \cdot a \cdot x)
\]
because in a labelled transition system, the behavior that can be observed after a \( \delta \) transition must be observable also after another \( \delta \) transition [22].

B. Comparing sets of required traces

It would be natural to guess if comparing sets of r-traces could be linked to comparing sets of \( \text{uioco} \) 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 IV.1.** Consider the specifications of figure 5 with input and output labels \( L_I = \{a, b\} \) and \( L_O = \{x, y\} \). 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 \( \text{uioco} \) (or \( \text{ioco}_{\text{Rtraces}(s)} \)) conforming to \( s \) is a subset of the set of implementations which are \( \text{uioco} \) conforming to \( s' \). The same reasoning can be done for \( s' \) and \( s'' \). Formally: \( \text{SUT}(s) \subseteq \text{SUT}(s') \subseteq \text{SUT}(s'') \).

![Figure 5: Three specifications whose sets of \( \text{uioco} \) conforming implementations are in a subset relation. Their sets of r-traces are not in the same kind of relation.](image)

We can easily find the sets of r-traces (note that sequences of multiple \( \delta \) have been reduced to a single \( \delta \)): 
\[
\text{Rtraces}(s) = \{\epsilon, \delta \cdot a, a \cdot a \cdot x, \delta \cdot a \cdot x\} \\
\text{Rtraces}(s') = \{\epsilon, \delta \cdot a, a \cdot a \cdot x, a \cdot y, \delta \cdot a \cdot y\} \\
\text{Rtraces}(s'') = \{\epsilon\}
\]
and so: \( \text{Rtraces}(s'') \subseteq \text{Rtraces}(s) \subseteq \text{Rtraces}(s') \).

From this example it is clear that we need more information in order to find a relation among specifications that is linked to their \( \text{uioco} \) conforming implementations, such as a kind of \( \text{uioco} \) 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 III.

C. Input-output Conformance between Specifications

Where, at the beginning of section III, \( \text{uioco} \) failed, \( \text{ioco}_{\text{Rtraces}(s)} \) could be successful, but \( \text{ioco}_{\text{Rtraces}(s)} \) 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 must be input-enabled. The following definitions allow us to check for input-output conformance on r-traces between specifications.

**Definition IV.5.** Let \( s \in \text{LTS}(L_I, L_O) \) be a specification and \( \sigma \in L_I^* \) be a trace,
\[
\text{Rin}(s \text{ after } \sigma) = \text{def} \{a \in \text{in}(s \text{ after } \sigma) \mid \exists \sigma' \in \text{Rtraces}(s) : \sigma \cdot a \preceq \sigma'\}
\]
Basically \( \text{Rin}(s \text{ after } \sigma) \) contains those input labels that are allowed in all the reachable states after observing \( \sigma \) and that still identify a (prefix of an) r-trace.

**Definition IV.6.** \( s \text{ wioco } s' \iff \forall \sigma \in \text{Rtraces}(s') : \\
1) \text{out}(s \text{ after } \sigma) \subseteq \text{out}(s' \text{ after } \sigma) \land \\
2) \text{Rin}(s \text{ after } \sigma) \supseteq \text{Rin}(s' \text{ after } \sigma)\)

As for \( \text{uioco} \) between specifications in the previous section, \( \text{wioco} \) adapts \( \text{ioco}_{\text{Rtraces}(s)} \) to non input-enabled implementations. We will show that the set of \( \text{uioco} \) conforming implementations of two specification in a \( \text{wioco} \) relation are in a subset relation and vice versa.

**Theorem IV.4.** Let \( s, s' \in \text{LTS}(L_I, L_O) \) be two specifications over the same input and output sets,
\[
s \text{ wioco } s' \iff \text{SUT}(s) \subseteq \text{SUT}(s')
\]

Theorems IV.4 gives 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 relation with \( s \).

V. REQUIRED TRACES AUTOMATA

A suspension automaton [21], or deterministic quiescent labelled transition system [1], is a model built from a specification that represents exactly the set of traces used to test for \( \text{ioco} \) conformance. It is obtained by adding \( \delta \)-loops for all quiescent states and then determining the obtained model. This approach does not work for \( \text{uioco} \)
or io\textsubscript{co}\textsubscript{\mu traces}(s) because the information on underspecified traces is lost during the determinization process.

**Example V.1.** Figure 6 shows a specification \( s \in \text{LTS}\{\{a\}, \{x\}\} \) to which \( \delta \)-loops have been added (figure 6a). The trace \( a\cdot a \) is underspecified because of the left branch, thus, in an \( \text{uioco} \) conforming implementation, any behavior is allowed after \( a\cdot a \). This information is lost after the determinization of \( s \) (figure 6b).

However we can first make explicit the underspecifications, e.g., using demonic completion (figure 7a), and then apply the same steps used to construct the suspension automaton, i.e., adding \( \delta \)-transitions 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 \( \chi \), i.e. a state that allows any behavior.

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 \( \delta \) and \( \tau \) transitions. The set of \( r \)-traces is not prefix closed so we need the concept of accepting state in order to be able to describe it with a model. Note that states that are trace equivalent to \( \chi \) 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 \( \chi \). 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 \( \text{UiocoSpec}(s) \), because we still need to handle the special cases in which a state with a \( \delta \)-loop could be marked as accepting (see example V.2).

**Example V.2.** Figure 7 shows the demonic completion of the specification used in example V.1 and its determinization with accepting states. Figure 8a gives the automaton obtained by removing from the model of figure 7b those states that behave like the chaotic state \( \chi \), i.e., those states that, when they have been reached, allow any behavior.

The language accepted by this automaton still contains too many traces, for example \( \delta \) and \( a\cdot \delta \cdot \delta \cdot a \) are accepted, but they are not \( r \)-traces.

Applying the \( \delta \)-sequences optimization described in section IV-A allows us not only to remove \( \delta \)-sequences from the accepted language, but it will also solve the special case described above. We call the final automaton, obtained after the \( \delta \)-sequences optimization, the required traces automaton of \( s \), or RTA\((s)\).

**A. \( \delta \)-sequences optimization for required traces automata**

In figure 8 it is explicit why we need to manipulate the automaton obtained so far: the initial state, for example, is marked as accepting, in fact \( \epsilon \) is an \( r \)-trace. Unfortunately this will also add the trace \( \delta \) to the accepted language, because of the \( \delta \)-loop. Furthermore, as described in section IV-A, sequences of \( \delta \) must be reduced to a single \( \delta \). In order to consider these two cases we apply the following steps to the required traces automaton:

1) replace \( \delta \)-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 \( q_1 \xrightarrow{\lambda} q_2 \) with \( \lambda \neq \delta \);

2) replace \( \delta \)-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 \( q \xrightarrow{\lambda} q_1 \) with \( \lambda \neq \delta \);

*Figure 7: Demoniac completion of \( s \) and its determinization. State \( \chi_1 \) is not represented in the graphs. A double circle represents an accepting state.*

*Figure 8: After \( \delta \)-sequences optimization, the required traces automation represents the set of \( r \)-traces described in section IV*
Theorem V.1. Let $s \in \mathcal{LTS}(L_1, L_U)$ be a specification, $\sigma \in L_\delta$ be a trace and $\text{RTA}(s)$ be the required traces automaton of $s$, then

$$\sigma \in \text{Rtraces}(s) \iff \sigma \text{ is accepted by } \text{RTA}(s)$$

B. The algorithm

The required traces automaton of a specification $s$, can be constructed using algorithm 1. The algorithm applies

Algorithm 1 Find the required traces automaton of $s \in \mathcal{LTS}(L_1, L_U)$

1: add loops $s \overset{\delta}{\to} s$ for all quiescent states
2: build $DC(s)$, the demonic completion of $s$
3: determinize $DC(s)$ obtaining a new input-output transition system $DC^{\text{out}}(s) = (Q, L_1, L_U, T, q_0)$
4: for each $q \in Q$ do
5:   if $(\text{out}(q) \neq L_U \cup \delta) \land (\exists p \ni p \overset{\lambda}{\to} q \land \lambda \neq \delta \land \lambda \neq \tau)$ then
6:     mark $q$ as accepting
7:   for each $q \in Q$ do
8:     if $q$ is trace equivalent to chaos state $\chi$ then mark $q$ as chaotic
9:   remove all chaotic states from $Q$
10: for each $(q, \delta, q_1) \in T$ do
11:   add a new state $p$ to $Q$ ($\delta$ sequences optimization steps 1 and 2)
12:   add $(q, \delta, p)$ to $T$
13: for each $a \in L_1$ such that $q_1 \overset{a}{\to} q_2$ do
14:   add $(p, a, q_2)$ to $T$
15: remove $(q, \delta, q_1)$ from $T$

the steps described in the previous sections: lines 1 and 2 build the demonic completion (with $\delta$ transitions) of the initial specification $s$, making explicit the underspecification implied by $\text{uioco}$ relation. Line 3 determines 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 any output actions as accepting. The language accepted by the obtained automaton is not yet the set of $\text{r-traces}$ due to possible $\delta$-sequences. For this reason we first remove states that are trace equivalent to $\chi$ at lines 7 and 8 and then we apply the $\delta$-sequences optimization from line 10 to 14.

VI. SPECIFICATION SELECTION AS A TEST SELECTION METHOD

The set of $\text{r-traces}$ is still usually an infinite set, even after $\delta$-optimization. In checking the conformance of an implementation with a given specification we can only use some test cases, i.e. some $\text{r-traces}$, 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 $\text{r-traces}$ 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 $\text{uioco}$ conformance between the implementation and $s'$.

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

Definition VI.1. Given $s \in \mathcal{LTS}(L_1, L_U)$ and a finite set $R \subseteq \text{Rtraces}(s)$, then $s_R \in \mathcal{LTS}(L_1, L_U)$ is a specification such that $\text{Rtraces}(s_R) = R$ and $\forall \sigma \in R : \text{out}(s \text{ after } \sigma) \subseteq \text{out}(s_R \text{ after } \sigma)$.

Such specification exists (theorem VI.1) and $s$ is $\text{wioco}$ conforming to it (theorem VI.2).

Theorem VI.1. Given $s \in \mathcal{LTS}(L_1, L_U)$ and a finite set $R \subseteq \text{Rtraces}(s)$, then a specification $s_R$ as defined in definition VI.1 exists.

Theorem VI.2. Given $s \in \mathcal{LTS}(L_1, L_U)$, a finite set $R \subseteq \text{Rtraces}(s)$ and $s_R$ as defined in definition VI.1, then $s$ $\text{wioco}$ $s_R$.

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 $\text{Rtraces}(s)$ or by modifying the specification $s$ by applying some transformations that do not remove any implementation from $\text{SUT}(s)$.

Following the first method, it is possible to provide heuristics that can be used to select traces among $\text{r-traces}$ 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 [20] 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). All these transformations, if done under certain conditions, do not reduce the set $SUT(s)$. 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 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.

VII. EXAMPLE

Let us give an example of the concepts that have been introduced in sections V and VI. The specification $s$ of figure 9a represents a vending machine. The only input is $b$ (pressing a button) and the possible outputs are $c$ and $t$ (coffee and tea). In the figure the absence of output, i.e. quiescence, is not explicit. 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 that 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.

Figure 9b represents the required traces automaton of $s$. The set of r-traces induced by that automaton is infinite, due to three transitions going back to the initial state.

We can remove (or add) some transitions from $s$ and obtain a new specification $s'$ such that $s \textbf{wioco} s'$. Such a specification can be the one of figure 10a.

We removed two input transitions from the bottom left state and we added an output transition to the bottom right state. Being $SUT(s) \subseteq SUT(s')$ we can test an implementation for $\textbf{uioco}$ conformance. If it is not conforming to $s'$ then it is not conforming to $s$ as well, i.e. it is sound. However, its set of r-traces is still infinite, see figure 10b.

Unfolding the cycles in $s$ we can obtain another weaker specification $s''$. An example of $s''$ is given in figure 11a. Also in this case it is valid that $s \textbf{wioco} s''$.

The required traces automaton of $s''$ is a finite directed acyclic graph (see figure 11b), thus the set of r-traces is finite. An implementation can be exhaustively tested with respect to $s''$.

In this example we have defined two specifications which are $\textbf{wioco}$ related to the initial one, providing a kind of test selection based on specifications rather than on test cases. A specification can be selected in a way such that the amount of test cases to be run is smaller. Nevertheless it is also possible to first reduce the test cases to a smaller set. This subset represents a new, weaker specification with respect to which we can test conformance.

VIII. CONCLUSION

In this work we have addressed two problems that arise while dealing with test selection for the $\textbf{ioco}$ 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.

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 also used
this new set of traces to define a new relation between specifications that can also be used as implementation relation for testing non-input enabled implementations.

We also 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 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 \textit{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 VI.

REFERENCES


