Decomposability in Input Output Conformance Testing

Neda Noroozi
Eindhoven University of Technology
Eindhoven, The Netherlands
n.noroozi@tue.nl

Mohammad Reza Mousavi
Eindhoven University of Technology
Eindhoven, The Netherlands
Center for Research on Embedded Systems (CERES)
Halmstad University, Sweden
m.r.mousavi@tue.nl

Tim A.C. Willemse
Eindhoven University of Technology
Eindhoven, The Netherlands
t.a.c.willemse@tue.nl

We study the problem of deriving a specification for a third-party component, based on the specification of the system and the environment in which the component is supposed to reside. Particularly, we are interested in using component specifications for conformance testing of black-box components, using the theory of input-output conformance (ioco) testing. We propose and prove sufficient criteria for decompositionality, i.e., that components conforming to the derived specification will always compose to produce a correct system with respect to the system specification. We also study the criteria for strong decomposability, by which we can ensure that only those components conforming to the derived specification can lead to a correct system.

1 Introduction

Enabling reuse and managing complexity are among the major benefits of using compositional approaches in software and systems engineering. This idea has been extensively adopted in several different subareas of software engineering, such as product-line software engineering. One of the cornerstones of the product-line approach is to reuse a common platform to build different products. This common platform should ideally comprise different types of artifacts, including test-cases, that can be re-used for various products of a given line. In this paper, we propose an approach to conformance testing, which allows to use a high-level specification and derive specifications for to-be-developed components (or subsystems) given the platform on which they are to be deployed. We call this approach *decompositional* testing and refer to the process of deriving specifications as *quotienting* (inspired by its counterpart in the domain of formal verification).

We develop our approach within the context of input-output conformance testing (**ioco**) [13], a model-based testing theory using formal models based on input-output labeled transition systems (IOLTSs). An implementation i is said to conform to a specification s, denoted by i **ioco** s, when after each trace in the specification, the outputs of the implementation are among those prescribed by the specifications.

For a given platform (environment) $\bar{e}$, whose behavior is given as an IOLTS, a quotient of a specification $\bar{s}$ by the platform $\bar{e}$, denoted by $\bar{s}_{/\bar{e}}$, is the specification that describes the system after filtering out the effect of $\bar{e}$. The structure of a system consisting of $\bar{e}$ and unknown component $\bar{c}$ is represented in Figure 1, whose behavior is described by a given specification $\bar{s}$. We would like to construct $\bar{s}_{/\bar{e}}$ such that it captures the behavior of any component $\bar{c}$ which, when deployed on $\bar{e}$ (put in parallel and possibly synchronize with $\bar{e}$) conforms to $\bar{s}$. Put formally, $\bar{s}_{/\bar{e}}$ is the specification which satisfies the following bi-implication:

A. Petrenko, H. Schlingloff (Eds.): Eighth Workshop on
Model-Based Testing (MBT 2013)
EPTCS ??, 2013, pp. 16–31, doi:10.4204/EPTCS.??.2

$$\forall \bar{c}, \bar{e}.\ \ \bar{c}\ \mathbf{ioco}\ \bar{s}_{/\bar{e}} \quad \Leftrightarrow \quad \bar{c} \| \bar{e}\ \mathbf{ioco}\ \bar{s}$$

The criteria for the implication from left to right, which is essential for our approach, are called *decomposability*. The criteria for the implication from right to left guarantee that quotienting produces the precise specification for the component and is called *strong decomposability*. We study both criteria in the remainder of this paper. Moreover, we show that strong decomposability can be combined with *on-the-fly* testing, thereby avoiding constructing the witness to the decomposability explicitly upfront.

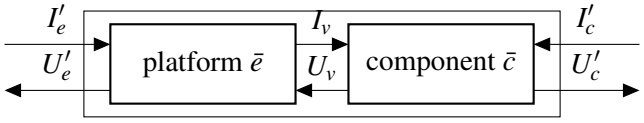


Figure 1: Strucure of a system composed of platform $\bar{e}$ and component $\bar{c}$ whose behavior is defined by a given specification $\bar{s}$. The language of platform $\bar{e}$ comprises $(I'_e \cup U_v) \cup (U'_e \cup I_v)$. Similarly, $(I'_c \cup I_v) \cup (U'_c \cup U_v)$ is the language of component $\bar{c}$. The platform $\bar{e}$ and component $\bar{c}$ interface via I_v and U_v which are hidden from the viewpoint of an external observer.

Related Work. The study of compositional and modular verification for various temporal and modal logics has attracted considerable attention and several compositional verification techniques have been proposed for such logics; see, e.g., [2, 7, 10, 6]. Decompositional reasoning aims at automatically decomposing the global property to be model checked into local properties of (possibly unknown) components, a technique that is often called quotienting. The notion of quotient introduced in the present paper is inspired by its corresponding notion in the area of (de)compositional model-checking, and is substantially adapted to the setting for input-output conformance testing, e.g., by catering for the distinction between input and output actions and taking care of (relative) quiescence of components. In the area of model-based testing, we are aware of a few studies dedicated to the issue of (de)composition [3, 5, 14], of which we give an overview below.

In [3] the compositionality of the **ioco**-based testing theory is investigated. Assuming that implementations of components conform to their specifications, the authors investigate whether the composition of these implementations still conforms to the composition of the specifications. They show that this is not necessarily the case and they establish conditions under which **ioco** is a compositional testing relation.

In [5], Frantzen and Tretmans study when successful integration of components by composing them in certain ways can be achieved. Successful integration is determined by two conditions: the integrated system correctly provides services, and interaction with other components is proper. For the former, a specification of the provided services of the component is assumed. Based on the **ioco**-relation, the authors introduce a new implementation relation called **eco**, which allows for checking whether a component conforms to its specification as well as whether it uses other components correctly. In addition, they also propose a bottom-up strategy for building an integrated systems.

Another problem closely related to the problem we consider in this paper is *testing in context*, also known as *embedded testing* [14]. In this setting, the system under test comprises a component $\bar{c}$ which is embedded in a context $\bar{u}$. Component $\bar{c}$ is isolated from the environment and all its interactions proceed through $\bar{u}$ (which is assumed to be correctly implemented). The implementation $\bar{i}$ and specification $\bar{s}$ of the system composed of $\bar{u}$ and $\bar{c}$, are assumed to be available. The problem of testing in context then entails generating a test suite that allows for detecting incorrect implementations $\bar{i}$ of component $\bar{c}$.

Although testing in context and decomposability share many characteristics, there are key differences between the two. We do not restrict ourselves to embedded components, nor do we assume the platforms to be fault-free. Contrary to the testing in context approach, decomposing a monolithic specification is the primary challenge in our work; testing in context already assumes the specification is the result of a composition of two specifications. Moreover, in testing in context, the component $\bar{c}$ is tested through context $\bar{u}$ whereas our approach allows for testing the component directly through its deduced specification. As a result, we do not require that the context is always available while testing the component, which is particularly important in case the platform is a costly resource.

For similar reasons, asynchronous testing [11, 8, 15], which can be considered as some form of embedded testing, is different from the work we present in this paper.

Structure. We give a cursory overview of **ioco**-based formal testing in Section 2. The notions of decomposability and strong decomposability are formalized in Section 3. We present sufficient conditions for determining whether a given specification is decomposable in Section 4 and whether it is strongly decomposable in Section 5. We conclude in Section 6. ***Additional examples and results, together with all proofs for the lemmata and theorems can be found in [9].***

2 Preliminaries

Conformance testing is about checking that the observable behavior of the system under test is included in the prescribed behavior of the specification. In order to formally reason about conformance testing, we need a model for reasoning about the behaviors described by a specification, and assume that we have a formal model representing the behaviors of our implementations, so that we can reason about their conformance mathematically.

In this paper, we use variants of the well-known Labeled Transition Systems as a behavioral model for both the specification and the system under test. The Labeled Transition System model assumes that systems can be represented using a set of states and transitions, labeled with events or *actions*, between such states. A tester can observe the events leading to new states, but she cannot observe the states. We assume the presence of a special action τ, which we assume is unobservable to the tester.

Definition 1 (IOLTS) *An input-output labeled transition system (IOLTS) is a tuple $\langle S, I, U, \rightarrow, \bar{s}\rangle$, where S is a set of states, I and U are disjoint sets of observable* inputs *and* outputs*, respectively, $\rightarrow \subseteq S \times (I \cup U \cup \{\tau\}) \times S$ is the transition relation (we assume $\tau \notin I \cup U$), and $\bar{s} \in S$ is the initial state. The class of IOLTSs ranging over inputs I and outputs U is denoted* $\mathsf{IOLTS}(I,U)$.

Throughout this section, we assume an arbitrary, fixed IOLTS $\langle S, I, U, \rightarrow, \bar{s}\rangle$, and we refer to this IOLTS by referring to its initial state $\bar{s}$. We write L for the set $I \cup U$. Let $s, s' \in S$ and $x \in L \cup \{\tau\}$. In line with common practice, we write $s \xrightarrow{x} s'$ rather than $(s,x,s') \in \rightarrow$. Furthermore, we write $s \xrightarrow{x}$ whenever $s \xrightarrow{x} s'$ for some $s' \in S$, and $s \not\xrightarrow{x}$ when not $s \xrightarrow{x}$. A *word* is a sequence over the input and output symbols. The set of all words over L is denoted L^*, and ε is the empty word. For words $\sigma, \rho \in L^*$, we denote the concatenation of σ and ρ by $\sigma\rho$. The transition relation is generalized to a relation over words by the following deduction rules:

$$\frac{}{s \overset{\varepsilon}{\Longrightarrow} s} \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \qquad s'' \xrightarrow{x} s' \qquad x \neq \tau}{s \overset{\sigma x}{\Longrightarrow} s'} \qquad \frac{s \overset{\sigma}{\Longrightarrow} s'' \qquad s'' \xrightarrow{\tau} s'}{s \overset{\sigma}{\Longrightarrow} s'}$$

We adopt the notational conventions we introduced for $\rightarrow$ for $\Longrightarrow$. A state in the IOLTS $\bar{s}$ is said to *diverge* if it is the source of an infinite sequence of τ-labeled transitions. The IOLTS $\bar{s}$ is *divergent* if one of its reachable states diverges. Throughout this paper, we confine ourselves to non-divergent IOLTSs.

Definition 2 *Let $s' \in S$ and $S' \subseteq S$. The set of* traces*,* enabled actions *and* weakly enabled actions *for s and S' are defined as follows:*

- $\mathsf{traces}(s) = \{\sigma \in L^* \mid s \stackrel{\sigma}{\Rightarrow}\}$, *and* $\mathsf{traces}(S') = \bigcup_{s' \in S'} \mathsf{traces}(s')$.
- $\mathsf{init}(s) = \{x \in L \cup \{\tau\} \mid s \stackrel{x}{\rightarrow}\}$, *and* $\mathsf{init}(S') = \bigcup_{s' \in S'} \mathsf{init}(s')$.
- $\mathsf{Sinit}(s) = \{x \in L \mid s \stackrel{x}{\Rightarrow}\}$, *and* $\mathsf{Sinit}(S') = \bigcup_{s' \in S'} \mathsf{Sinit}(s')$.

Quiescence and Suspension Traces. Testers often not only have the power to observe events produced by an implementation, they can also observe the *absence* of events, or *quiescence* [13]. A state $s \in S$ is said to be *quiescent* if it does not produce outputs and it is *stable*. That is, it cannot, through internal computations, evolve to a state that *is* capable of producing outputs. Formally, state s is quiescent, denoted $\delta(s)$, whenever $\mathsf{init}(s) \subseteq I$. In order to formally reason about the observations of inputs, outputs and quiescence, we introduce the set of *suspension traces*. To this end, we first generalize the transition relation over words to a transition relation over suspension words. Let L_δ denote the set $L \cup \{\delta\}$.

$$\frac{s \stackrel{\sigma}{\Longrightarrow} s'}{s \stackrel{\sigma}{\Longrightarrow}_\delta s'} \qquad \frac{\delta(s)}{s \stackrel{\delta}{\Rightarrow}_\delta s} \qquad \frac{s \stackrel{\sigma}{\Longrightarrow}_\delta s'' \qquad s'' \stackrel{\rho}{\Longrightarrow}_\delta s'}{s \stackrel{\sigma\rho}{\Longrightarrow}_\delta s'}$$

The following definition formalizes the set of suspension traces.

Definition 3 *Let $s \in S$ and $S' \subseteq S$. The set of* suspension traces *for s, denoted $\mathsf{Straces}(s)$ is defined as the set $\{\sigma \in L_\delta^* \mid s \stackrel{\sigma}{\Rightarrow}_\delta\}$; we set $\mathsf{Straces}(S') = \bigcup_{s' \in S'} \mathsf{Straces}(s')$.*

Input-Output Conformance Testing with Quiescence. Tretmans' **ioco** testing theory [13] formalizes black box conformance of implementations. It assumes that the behavior of implementations can always be described adequately using a class of IOLTSs, called *input output transition systems*; this assumption is the so-called *testing hypothesis*. Input output transition systems are essentially plain IOLTSs with the additional assumption that inputs can always be accepted.

Definition 4 (IOTS) *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be an IOLTS. A state $s \in S$ is* input-enabled *iff $I \subseteq \mathsf{Sinit}(s)$; the IOLTS $\bar{s}$ is an* input output transition system *(IOTS) iff every state $s \in S$ is input-enabled. The class of input output transition systems ranging over inputs I and outputs U is denoted $\mathsf{IOTS}(I, U)$.*

While the **ioco** testing theory assumes input-enabled implementations, it does not impose this requirement on specifications. This facilitates testing using partial specifications, *i.e.*, specifications that are under-specified. We first introduce the main concepts that are used to define the family of conformance relations of the **ioco** testing theory.

Definition 5 *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be an IOLTS. Let $s \in S$, $S' \subseteq S$ and let $\sigma \in L_\delta^*$.*

- s after $\sigma = \{s' \in S \mid s \stackrel{\sigma}{\Rightarrow}_\delta s'\}$, *and* S' after $\sigma = \bigcup_{s' \in S'} s'$ after σ.
- $\mathsf{out}(s) = \{x \in L_\delta \setminus I \mid s \stackrel{x}{\Rightarrow}_\delta\}$, *and* $\mathsf{out}(S') = \bigcup_{s' \in S'} \mathsf{out}(s')$.

The family of conformance relations for **ioco** are then defined as follows, see also [13].

Definition 6 (ioco) *Let $\langle R, I, U, \rightarrow, \bar{r}\rangle$ be an IOTS representing a realization of a system, and let IOLTS $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be a specification. Let $F \subseteq L_\delta^*$. We say that $\bar{r}$ is* input output conform *with specification $\bar{s}$, denoted $\bar{r}$ $\mathbf{ioco}_F$ $\bar{s}$, iff*

$$\forall \sigma \in F : \ \mathsf{out}(\bar{r} \text{ after } \sigma) \subseteq \mathsf{out}(\bar{s} \text{ after } \sigma)$$

The $\mathbf{ioco}_F$ conformance relation can be specialized by choosing an appropriate set F. For instance, in a setting with $F = \mathsf{Straces}(s)$, we obtain the **ioco** relation originally defined by Tretmans in [12]. The latter conformance relation is known to admit a sound and complete test case generation algorithm, see, *e.g.*, [12, 13]. Soundness means, intuitively, that the algorithm will never generate a test case that, when executed on an implementation, leads to a *fail* verdict if the test runs are in accordance with the specification. Completeness is more esoteric: if the implementation has a behavior that is not in line with the specification, then there is a test case that, in theory, has the capacity to detect that non-conformance.

Suspension automata. The original test case generation algorithm by Tretmans for the **ioco** relation relied on an automaton derived from an IOLTS specification. This automaton, called a *suspension automaton*, shares many of the characteristics of an IOLTS, except that the observations of quiescence are encoded explicitly as outputs: δ is treated as an ordinary action label which can appear on a transition. In addition, Tretmans assumes these suspension automata to be *deterministic*: any word that could be produced by an automaton leads to exactly one state in the automaton.

Definition 7 (Suspension automaton) *A suspension automaton(SA) is a deterministic IOLTS $\langle S, I, U \cup \{\delta\}, \rightarrow, \bar{s}\rangle$; that is, for all $s \in S$ and all $\sigma \in L^*$, we have $|s \text{ after } \sigma| \leq 1$.*

Note that determinism implies the absence of τ transitions. In [12], a transformation from ordinary IOLTSs to suspension automata is presented; the transformation ensures that trace-based testing using the resulting suspension automaton is exactly as powerful as **ioco**-based testing using the original IOLTS.

The transformation is essentially based on the subset construction for determinizing automata. Given an IOLTS, the transformation Δ defined below converts any IOLTS into an SA.

Definition 8 *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle \in \mathsf{IOLTS}(I, U)$. The SA $\Delta(\bar{s}) = \langle Q, I, U \cup \{\delta\}, \rightarrow, \bar{q}\rangle$ is defined as:*

- $Q = \mathbb{P}(S) \setminus \{\emptyset\}$.
- $\bar{q} = \bar{s}$ after ε.
- $\rightarrow \subseteq Q \times L_\delta \times Q$ *is the least relation satisfying:*

$$\frac{x \in L \qquad q \in Q}{q \stackrel{x}{\rightarrow} \{s' \in S \mid \exists s \in q \bullet s \stackrel{x}{\Rightarrow} s'\}} \qquad \frac{q \in Q}{q \stackrel{\delta}{\rightarrow} \{s \in q \mid \delta(s)\}}$$

Example 1 *Consider the IOLTS $\bar{s}$ depicted in Figure 2 on page 22. The IOLTS $\bar{s}$ is a specification of a malfunctioning vending machine which sells tea for one euro coin (c). After receiving money, it either delivers tea (t), refunds the money (r) or does nothing. Its suspension automaton $\Delta(\bar{s})$, with initial state $\bar{q}$, is depicted next to it. Note that the suspension traces of $\bar{s}$ and the traces of suspension automaton $\Delta(\bar{s})$ are identical.*

In general, a suspension automaton may not represent an actual IOLTS; for instance, in an arbitrary suspension automaton, it is allowed to observe quiescence, followed by a proper output. This cannot happen in an IOLTS. In [16], the set of suspension automata is characterized for which a transformation to an IOLTS *is* possible. Such suspension automata are called *valid*. Proposition 1 of [16] states that for any IOLTS $\bar{s}$, the suspension automaton $\Delta(\bar{s})$ is valid. Conversely, Theorem 2 of [16] states that any valid suspension automaton has the same testing power (with respect to **ioco**) as *some* IOLTS. This essentially means that the class of valid suspension automata can be used safely for testing purposes.

Parallel Composition. A software or hardware system is usually composed of subunits and modules that work in an orchestrated fashion to achieve the desired overall behavior of the software or hardware system. In our setting, we can formalize such compositions using a special operator $\_\|\_$ on IOLTSs: two IOLTSs can interact by connecting the outputs sent by one IOLTS to the inputs of the other IOLTS. We assume that such inputs and outputs are taken from a shared alphabet of actions. For the non-common actions the behavior of both IOLTSs is interleaved.

Definition 9 (parallel composition) *Let $\langle S_1, I_1, U_1, \rightarrow_1, \bar{s}_1\rangle$ and $\langle S_2, I_2, U_2, \rightarrow_2, \bar{s}_2\rangle$ be two IOLTSs with disjoint sets of input labels I_1 and I_2, and disjoint sets of output labels U_1 and U_2. The parallel composition of $\bar{s}_1$ and $\bar{s}_2$, denoted $\bar{s}_1 \| \bar{s}_2$ is the IOLTS $\langle Q, I, U, \rightarrow, \bar{s}_1 \| \bar{s}_2\rangle$, where:*

- $Q = \{s_1 \| s_2 \mid s_1 \in S_1, s_2 \in S_2\}$.
- $I = (I_1 \cup I_2) \setminus (U_1 \cup U_2)$ *and* $U = U_1 \cup U_2$.
- $\rightarrow \subseteq Q \times (L \cup \{\tau\}) \times Q$ *is the least relation satisfying:*

$$\frac{s_1 \xrightarrow{x}_1 s_1' \qquad x \notin L_2}{s_1 \| s_2 \xrightarrow{x} s_1' \| s_2} \qquad \frac{s_2 \xrightarrow{x}_2 s_2' \qquad x \notin L_1}{s_1 \| s_2 \xrightarrow{x} s_1 \| s_2'}$$

$$\frac{s_1 \xrightarrow{x}_1 s_1' \qquad s_2 \xrightarrow{x}_2 s_2' \qquad x \neq \tau}{s_1 \| s_2 \xrightarrow{x} s_1' \| s_2'}$$

The interaction *between* components is typically intended to be unobservable by a tester. This is not enforced by the parallel composition, but can be specified by combining parallel composition with a *hiding* operator, which is formalized below.

Definition 10 (hiding) *Let $\langle S, I, U, \rightarrow, \bar{s}\rangle$ be an IOLTS, and let $V \subseteq U$. The IOLTS resulting from hiding events from the set V, denoted by* $\mathbf{hide}[V]\,\mathbf{in}\, s$ *is the IOLTS $\langle S, I, U \setminus V, \rightarrow', \bar{s}\rangle$, where $\rightarrow'$ is defined as the least relation satisfying:*

$$\frac{s \xrightarrow{x} s' \qquad x \notin V}{\mathbf{hide}[V]\,\mathbf{in}\, s \xrightarrow{x}{}' \mathbf{hide}[V]\,\mathbf{in}\, s'} \qquad \frac{s \xrightarrow{x} s' \qquad x \in V}{\mathbf{hide}[V]\,\mathbf{in}\, s \xrightarrow{\tau}{}' \mathbf{hide}[V]\,\mathbf{in}\, s'}$$

Note that the hiding operator may turn non-divergent IOLTSs into divergent IOLTSs. As divergence is excluded from the **ioco** testing theory, we must assume such divergences are not induced by composing two implementations in parallel and hiding all successful communications. Since implementations are assumed to be input enabled, this can only be ensured whenever components that are put in parallel never produce infinite, uninterrupted runs of outputs over their alphabet of shared output actions. Implementations adhering to these constraints are referred to as *shared output bounded* implementations. From hereon, we assume that all the implementions considered are shared output bounded.

3 Decomposibility

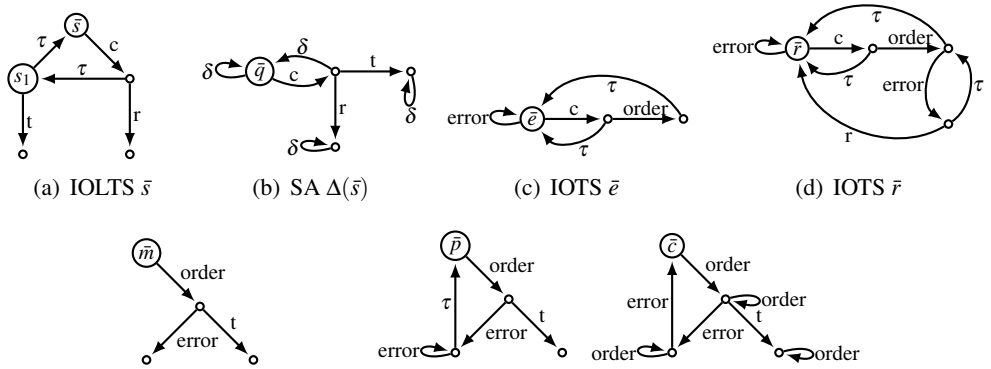
Software can be constructed by decomposing a specification of the software in specifications of smaller complexity. Reuse of readily available and well-understood platforms or environments can steer such a decomposition. Given the prevalence of such platforms, the software engineering and associated testing problem thus shifts to finding a proper specification of the system from which the platform behavior has been factored out. Whether this is possible, however, depends on the specification; if so, we say that a specification is *decomposable*.

The decomposability problem requires known action alphabets for both the specification and the platform. Hence, we first fix these alphabets and illustrate how these are related. Hereafter, L_s denotes the action alphabet of the specification $\bar{s}$ and L_e denotes the action alphabet of the platform $\bar{e}$. The actions of L_e not exposed to $\bar{s}$ are contained in action alphabet L_v, *i.e.*, we have $L_v = L_e \setminus L_s$. The action alphabet of the quotient will be denoted by L, *i.e.* $L = (L_s \setminus L_e) \cup L_v$. The relation between the above alphabets is illustrated in Figure 1 in the introduction.

Definition 11 (Decomposability) *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification, and let $\bar{e} \in \mathsf{IOTS}(I_e, U_e)$ be an implementation. Let $L_v = I_v \cup U_v$ be a set of actions of $\bar{e}$ not part of $\bar{s}$. Specification $\bar{s}$ is said to be* decomposable *for IOTS $\bar{e}$ iff there is some specification $\bar{s}' \in \mathsf{IOLTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)$ for which both:*

- $\exists \bar{c} \in \mathsf{IOTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v) \bullet \bar{c}$ ***ioco*** $\bar{s}'$*, and*
- $\forall \bar{c} \in \mathsf{IOTS}((I_e \setminus I_e) \cup I_v, (U_e \setminus U_e) \cup U_v) \bullet \bar{c}$ ***ioco*** $\bar{s}' \Longrightarrow$ **hide**$[L_v]$ **in** $\bar{c} \| \bar{e}$ ***ioco*** $\bar{s}$

Decomposability of a specification $\bar{s}$ essentially ensures that a specification $\bar{s}'$ for a subcomponent exists that guarantees that every **ioco**-correct implementation of it is also guaranteed to work correctly in combination with the platform.



(a) IOLTS $\bar{s}$ (b) SA $\Delta(\bar{s})$ (c) IOTS $\bar{e}$ (d) IOTS $\bar{r}$

(e) IOLTS $\bar{m}$ (f) IOLTS $\bar{p}$ (g) IOTS $\bar{c}$

Figure 2: A specification of a vending machine ($\bar{s}$), two behavioral models of an implemented money component ($\bar{e}$ and $\bar{r}$) and two specifications for a drink component ($\bar{m}$ and $\bar{p}$) with the behavioral model of an implementation of the drink component ($\bar{c}$).

Example 2 *Consider IOLTSs depicted in Figure 2. The IOTS $\bar{e}$ 2(c) presents the behavioral model of an environment which after receiving a coin (c) either orders drink (order) or does nothing. Upon receiving an error signal (error), never refunds the money (r). Component $\bar{e}$ interacts with another component*

through actions 'order' and 'error'; together, the components implement a vending machine for which IOLTS $\bar{s}$ 2(a) is the specification. The IOLTS $\bar{m}$ 2(e) is a specification of a drink component which delivers tea after receiving a drink order. If it encounters a problem in delivering the drink, it signals an error. Specification $\bar{m}$ guarantees that the combination of component $\bar{e}$ with any drink component implementation conforming to $\bar{m}$, also conforms to $\bar{s}$.

It may, however, be the case that an implementation, in combination with a given platform, perfectly adheres to the overall specification $\bar{s}$, and, yet fails to pass the conformance test for $\bar{s}'$. As a consequence, non-conformance of an implementation to $\bar{s}'$ may not by itself be a reason to reject the implementation.

Example 3 *Consider IOLTSs in Figure 2. The IOLTS $\bar{m}$ 2(e) is a witness for decomposability of IOLTS $\bar{s}$2(a) for platform $\bar{e}$2(c). Thus, any compound system built of IOTS $\bar{e}$ and a component conforming to $\bar{m}$ is guaranteed to be in conformance with IOLTS $\bar{s}$. Now, consider IOTS $\bar{c}$ 2(g) which incorrectly implements the functionality specified in IOLTS $\bar{m}$ 2(e), as it sends 'error' twice. Observe that, nevertheless,* $\mathbf{hide}[\{error, order\}]\,\mathbf{in}\,\bar{c}\|\bar{e}$ *still conforms to $\bar{s}$.*

It is often desirable to consider specifications $\bar{s}'$ for which one *only* has to check whether an implementation $\bar{c}$ adheres to $\bar{s}'$, *i.e.*, specifications for which it is guaranteed that a failure of an implementation $\bar{c}$ to comply to $\bar{s}'$ also guarantees that the combination $\bar{c}\|\bar{e}$ will violate the original specification $\bar{s}$. We can obtain this by considering a stronger notion of decomposability.

Definition 12 (Strong Decomposability) *Let $\bar{s} \in \mathsf{IOLTS}(I,U)$ be a specification, and let $\bar{e} \in \mathsf{IOTS}(I_e,U_e)$ be an implementation. Let $L_v = I_v \cup U_v$ be a set of actions of $\bar{e}$ not part of $\bar{s}$. Specification $\bar{s}$ is said to be* strongly decomposable *for IOTS $\bar{e}$ iff there is some specification $\bar{s}' \in \mathsf{IOLTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)$ for which both:*

- $\exists \bar{c} \in \mathsf{IOTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)\ \bullet\, \bar{c}\ \boldsymbol{ioco}\ \bar{s}'$, *and*
- $\forall \bar{c} \in \mathsf{IOTS}((I_s \setminus I_e) \cup I_v, (U_s \setminus U_e) \cup U_v)\ \bullet\, \bar{c}\ \boldsymbol{ioco}\ \bar{s}' \iff \mathbf{hide}[L_v]\,\mathbf{in}\,\bar{c}\|\bar{e}\ \boldsymbol{ioco}\ \bar{s}$

Example 4 *Consider the IOLTSs $\bar{p}$ and $\bar{e}$ in Figure 2; specification $\bar{p}$ is such that the combination of component $\bar{e}$ with any* shared output bounded *component that does not conform to $\bar{p}$, fails to comply to $\bar{s}$.*

4 Sufficient Conditions for Decomposibility

Checking whether a given specification is decomposable is a difficult problem. However, knowing that a specification is decomposable in itself hardly helps a design engineer. Apart from the question whether a specification is decomposable, one is typically interested in a witness for the decomposed specification, or *quotient*. Our approach to the decomposability problem is therefore constructive: we define a quotient and we identify several conditions that ensure that the quotient we define is a witness for the decomposability of a given specification.

One of the problems that may prevent a specification from being decomposable for a given platform $\bar{e}$ is that the latter may exhibit some behavior which unavoidably violates the specification $\bar{s}$. We shall therefore only consider platforms for which such violations are not present. We formalize this by checking whether the behavior of $\bar{e}$ is *included* in the behavior of $\bar{s}$; that is, we give conditions that ensure that $\bar{e}$ in itself cannot violate the given specification $\bar{s}$. Moreover, we assume that the input-enabled specification of $\bar{e}$ is available.

Assuming that the behavior of $\bar{e}$ is included in the behavior of the given specification $\bar{s}$, we then propose a quotient $\bar{s}'$ of $\bar{s}$ for $\bar{e}$ and prove sufficient conditions that guarantee that $\bar{s}$ is indeed decomposable and $\bar{s}'$ is a witness to that.

4.1 Inclusion relation

We say that the behavior of a given platform $\bar{e}$ is included in a specification $\bar{s}$ if the outputs allowed by $\bar{s}$ subsume all outputs that can be produced by $\bar{e}$. For this, we need to take possible communications between $\bar{e}$ and the to-be-derived quotient over the action alphabet L_v into account. Another issue is that we are dealing with two components, each of which may be quiescent. If component $\bar{e}$ is quiescent, its quiescence may be masked by outputs from the component with which it is supposed to interact. We must therefore consider a refined notion of quiescence. We say state s in specification $\bar{s}$ is *relatively quiescent* with respect to alphabet L_e, denoted by $\delta_{\bar{e}}(s)$, if s produces no output of L_e, *i.e.* $\mathsf{out}(s) \cap L_e = \emptyset$. Analogous to δ, the suspension traces of $\bar{s}$ can be enriched by adding the rule $s \stackrel{\delta_{\bar{s}}}{\Rightarrow}_{\delta} s$ for $\delta_{\bar{e}}(s)$ to be able to formally reason about the possibility of being relatively quiescent with respect to L_e. We write $\mathsf{Straces}_{\bar{e}}(\bar{s})$ to denote this enriched set of suspension traces of $\bar{s}$.

Since the suspension traces of $\bar{s}$ and $\bar{e}$ differ as a result of different alphabets, we introduce a *projection operator* which allows us to map the suspension traces of $\bar{s}$ to suspension traces of $\bar{e}$. The operator $_{-\downarrow_{L_e}}$ is defined as $(x\sigma)_{\downarrow_{L_e}} = x\sigma_{\downarrow_{L_e}}$ if $x \in L_e$; $(x\sigma)_{\downarrow_{L_e}} = \delta(\sigma_{\downarrow_{L_e}})$, if $x \in \{\delta, \delta_{\bar{e}}\}$; otherwise, $(x\sigma)_{\downarrow_{L_e}} = \sigma_{\downarrow_{L_e}}$.

Definition 13 *Let IOTS $\langle S_e, I_e, U_e, \rightarrow, \bar{e}\rangle$ be an implementation. Let IOLTS $\langle S_s, I_s, U_s, \rightarrow, \bar{s}\rangle$ be a specification. We say the behavior of $\bar{e}$ is included in $\bar{s}$, denoted by $\bar{e}$* **incl** *$\bar{s}$ iff*

$$\forall \sigma \in \mathsf{Straces}_{\bar{e}}(\bar{s}) : \mathsf{out}(\mathbf{hide}[L_v]\,\mathbf{in}\,\bar{e} \text{ after } \sigma_{\downarrow_{L_e}}) \subseteq \mathsf{out}(\bar{s} \text{ after } \sigma)$$

Example 5 *Consider the IOLTSs in Figure 2. We have $\bar{e}$* **incl** *$\bar{s}$. Consider the IOLTS $\bar{r}$ which has the same functionality with IOLTS $\bar{e}$ except that upon receiving an error signal (error), it may or may not refund the money (r). The behavior of $\bar{r}$ is not included in $\bar{s}$, because of observing the output r in $\bar{r}$ after executing $(ct)_{\downarrow_{L_e}}$ while $\bar{s}$ after execution of ct reaches to a quiescent state.*

4.2 Quotienting

We next focus on deriving a quotient of the specification $\bar{s}$, factoring out the behavior of the platform $\bar{e}$. A major source of complexity in defining such a quotient is the possible non-determinism that may be present in $\bar{s}$ and $\bar{e}$. We largely avert this complexity by utilizing the suspension automata underlying $\bar{s}$ and $\bar{e}$.

Another source of complexity is the fact that we must reason about the states of two systems running in parallel; such a system synchronizes on shared actions and interleaves on non-shared actions. We tame this conceptual complexity by formalizing an executes operator which, when executing a shared or non-shared action, keeps track of the set of reachable states for the (suspension automata) of $\bar{s}$ and $\bar{e}$. Formally, the executes operator is defined as follows.

Definition 14 *Let $\langle Q_s, I_s, U_s \cup \{\delta\}, \rightarrow_s, \bar{q}_s\rangle$ be a suspension automaton underlying specification IOLTS $\bar{s}$, and let $\langle Q_e, I_e, U \cup \{\delta\}, \rightarrow_e, \bar{q}_e\rangle$ be a suspension automaton underlying platform IOLTS $\bar{e}$. Let $q \in \mathbb{P}(Q_s \times Q_e)$ be a non-empty collection of sets and let $x \in L_s \setminus (L_e \setminus L_v)$.*

$$q \text{ executes } x = \begin{cases} \bigcup_{\sigma \in L_e^*} \bigcup_{(s,e) \in q} \{(q'_s, q'_e) \mid s \xrightarrow{\sigma}_s q'_s \text{ and } e \xrightarrow{\sigma x}_e q'_e\} & \text{if } x \in L_v \\ \bigcup_{\sigma \in L_e^*} \bigcup_{(s,e) \in q} \{(q'_s, q'_e) \mid s \xrightarrow{\sigma x}_s q'_s \text{ and } e \xrightarrow{\sigma}_e q'_e\} & \text{if } x \notin L_v \\ \bigcup_{\sigma \in L_e^*} \bigcup_{(s,e) \in q} \{(q'_s, q'_e) \mid s \xrightarrow{\sigma \delta}_s q'_s \text{ and } e \xrightarrow{\sigma \delta}_e q'_e\} & \text{if } x = \delta \end{cases}$$

Using the executes operator, we have an elegant construction of an automaton, called a *quotient automaton*, see below, which allows us to define sufficient conditions for establishing the decomposability of a given specification.

Definition 15 (Quotient Automaton) *Let $\langle Q_s, I_s, U_s \cup \{\delta\}, \rightarrow_s, \bar{q}_s \rangle$ be a suspension automaton underlying specification $\bar{s}$, and let $\langle Q_e, I_e, U_e \cup \{\delta\}, \rightarrow_e, \bar{q}_e \rangle$ be a suspension automaton underlying platform $\bar{e}$. The quotient of $\bar{s}$ by $\bar{e}$, denoted by $\bar{s}_{/\bar{e}}$ is a suspension automaton $\langle Q, I, U \cup \{\delta\}, \rightarrow, \bar{q} \rangle$ where:*

- *$Q = (\mathbb{P}(Q_s \times Q_e) \setminus \{\emptyset\}) \cup Q_\delta$, where $Q_\delta = \{q_\delta \mid q \in \mathbb{P}(Q_s \times Q_e), q \neq \emptyset\}$; for $q \notin Q_\delta$, we set $q^{-1} = q$ and for $q_\delta \in Q_\delta$, we set $q_\delta^{-1} = q$.*
- *$\bar{q} = \{(\bar{q}_s, \bar{q}_e)\}$.*
- *$I = (I_s \setminus I_e) \cup (U_e \setminus U_s)$ and $U = (U_s \setminus U_e) \cup \{\delta\} \cup (I_e \setminus I_s)$.*
- *$\rightarrow \subseteq Q \times L \times Q$ is the least set satisfying:*

$$\frac{a \in I \quad q^{-1} \text{ executes } a \neq \emptyset}{q \xrightarrow{a} q^{-1} \text{ executes } a} [I_1] \qquad \frac{x \in U_v \quad q \notin Q_\delta \quad q^{-1} \text{ executes } x \neq \emptyset}{q \xrightarrow{x} q^{-1} \text{ executes } x} [U_1]$$

$$\frac{x \in U \setminus U_v \quad \forall (s,e) \in q, \sigma \in \mathsf{traces}(s) \cap \mathsf{traces}(e) \cap (L_\delta^* \setminus L_\delta^* \delta) : \ x \in \mathsf{out}(s \text{ after } \sigma)}{q \xrightarrow{x} q^{-1} \text{ executes } x} [U_2]$$

$$\frac{\forall (s,e) \in q^{-1}, \sigma \in \mathsf{traces}(s) \cap \mathsf{traces}(e) : \ \delta \in \mathsf{out}(s \text{ after } \sigma)}{q \xrightarrow{\delta} q^{-1} \text{ executes } \delta} [\delta_1]$$

We briefly explain the construction of a quotient automaton. A *non-shared input action* is added to a state in the quotient automaton $\bar{s}_{/\bar{e}}$ if an execution of the corresponding state in $\bar{e}$ leads to a state in $\bar{s}$ at which that action is enabled (I_1, in combination with the second case in Definition 14). A *shared input action* obeys the same rule except that a state of $\bar{e}$ has to be reachable where that input action is taken (I_1, in combination with the first case in Definition 14). Note that a shared input action of $\bar{s}_{/\bar{e}}$ is an output action from the viewpoint of $\bar{e}$. In contrast, a *non-shared output action* is allowed at a state of $\bar{s}_{/\bar{e}}$ only if it is allowed by $\bar{s}$ after any possible execution of $\bar{e}$ (U_2) and a similar rule is applied to quiescence (δ_1). Analogous to the shared input actions, a *shared output action* is considered as an action of a state whenever a valid execution of the correspondent states in $\bar{e}$ leads to a state at which that output action is enabled (U_1). Because the shared actions are hidden in $\bar{s}$, a shared output action, in $\bar{s}_{/\bar{e}}$, may also be enabled at a state reached by δ transitions. Such a sequence of events is invalid due to the definition of quiescence. The observed problem is solved by adding a special set of states Q_δ to the states of the quotient automaton. These states represent quiescent states corresponding to the reachable states after executing δ in $\bar{s}_{/\bar{e}}$. Moreover, no shared output action is added to these states.

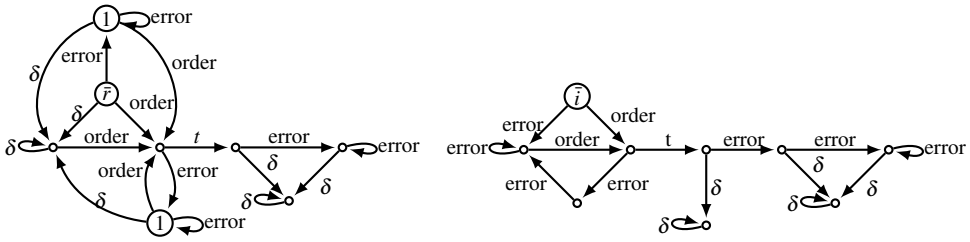


Figure 3: Two quotient automata derived using Definition 15

The quotient automaton derived from specification $\bar{s}$ and platform $\bar{e}$ is a suspension automaton: it is deterministic and it has explicit δ labels. Yet, the quotient automata we derive are not necessarily valid suspension automata. (As we recalled in Section 2, only *valid* suspension automata have the same testing power as ordinary IOLTSs.) We furthermore observe that there some quotient automata that are valid suspension automata but nevertheless only admit non-shared output bounded implementations as implementations that conform to the quotient. As observed earlier, such implementations unavoidably give rise to divergent systems when composed in parallel with the platform.

Example 6 *Consider SAs depicted in Figure 3, IOLTSs $\bar{s}$ and $\bar{e}$ in Figure 2 and IOLTS $\bar{l}$ derived by removing the internal transition from state s_1 to the initial state in $\bar{s}$. SA $\bar{r}$ is the quotient of $\bar{s}$ by $\bar{e}$. Likewise, SA $\bar{i}$ is the quotient of $\bar{l}$ by $\bar{e}$. Suspension automata $\bar{r}$ and $\bar{i}$ are valid SA regarding the definition of validity of suspension automata presented in [16] . Assume an arbitrary shared output bounded IOTS $\bar{c}$ whose length of the longest sequence on the shared output is n,* i.e. $\mathsf{out}(\bar{c}\ \mathsf{after}\ \sigma) \subseteq \{tea, \delta\}$ *for* $\sigma = \{error\}^n$. *Clearly, $\bar{c}$* ~~**ioco**~~ *$\bar{i}$, because* $\mathsf{out}(\bar{i}\ \mathsf{after}\ \sigma) = \{error\}$. *However, for any $n \geq 0$, there is always a shared output bounded IOTS that conforms to $\bar{r}$.*

In view of the above, we say that a quotient automaton is *valid* if it is a valid suspension automaton and strongly non-blocking.

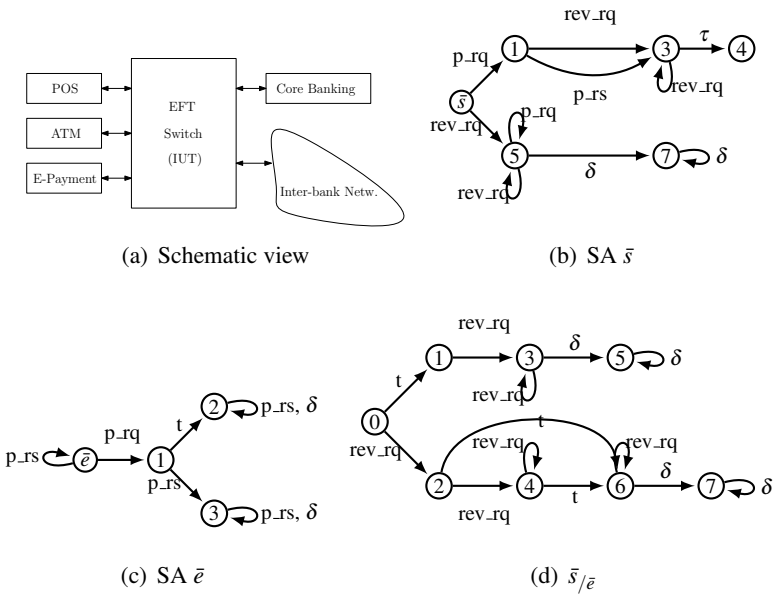
Definition 16 *Let $\bar{s}_{/\bar{e}}$ be a quotient automaton derived from a specification $\bar{s}$ and an environment $\bar{e}$. We say that $\bar{s}_{/\bar{e}}$ is* valid *iff both:*

- *$\bar{s}_{/\bar{e}}$ is a valid suspension automaton, and*
- *$\bar{s}_{/\bar{e}}$ is strongly non-blocking,* i.e. $\forall q \in \bar{s}_{/\bar{e}} \bullet \mathsf{out}(q) \cap ((U \setminus U_v) \cup \{\delta\}) \neq \emptyset$.

Strongly non-blocking ensures that the quotient automaton always admits a shared output bounded implementation that conforms to it. Furthermore, valid quotient automata are, by definition, also valid suspension automata. Since every valid suspension automaton underlies at least one IOLTS, we therefore have established a sufficient condition for the decomposability of a specification.

Theorem 1 *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification and let $\bar{e} \in \mathsf{IOTS}(I_e, U_e)$ be an environment. Then $\bar{s}$ is decomposable for $\bar{e}$ if $\bar{s}_{/\bar{e}}$ is a valid quotient automaton and $\bar{e}$* **incl** *$\bar{s}$.*

Note that the IOLTS underlying the quotient automaton is a witness to the decomposability of the specification; we thus not only have a sufficient condition for the decomposability of a specification but also a witness for the decomposition.



(a) Schematic view (b) SA $\bar{s}$

(c) SA $\bar{e}$ (d) $\bar{s}_{/\bar{e}}$

Figure 4: A schematic view of the EFT Switch, a suspension automata of simplified behavioral models of the EFT switch $\bar{s}$ and an implementation of the financial component $\bar{e}$, and the quotient of $\bar{s}$ w.r.t. $\bar{e}$

4.3 Example

To illustrate the notions introduced so far, we treat a simplified model of an Electronic Funds Transfer (EFT) switch, which we have studied and tested using ioco-based techniques [1]. A schematic view of this example is depicted in Figure 4(a). An EFT switch provides a communication mechanism among different components of a card-based financial system. On one side of the EFT switch, there are components, with which the end-user deals, such as Automated Teller Machines (ATMs), Point-of-Sale (POS) devices and e-Payment applications. On the other side, there are banking systems and the inter-bank network connecting the switches of different financial institutions.

The various involving parties in every transaction performed by an EFT switch in conjunction with the variety of financial transactions complicate the behavioral model of the EFT switch. Similar to any other complex software system, the EFT switch comprises many different components, some of which can be run individually.

A part of the simplified communication model of the EFT switch with a banking system in the purchase scenario is depicted in Figure 4(b). The scenario starts by receiving a purchase request from a POS; this initial part of the scenario is removed from the model, for the sake of brevity. Subsequently, the EFT switch sends a purchase request (p_rq) to the banking system. The EFT switch will reverse (rev_rq) the sent purchase request if the corresponding response (p_rs) is not received within a certain amount of time (e.g, an internal time-out occurs, denoted by τ). Due to possible delays in the network layer of the EFT switch, an external observer (tester) may observe the reverse request of a purchase even before the purchase request which is pictured in Fig 4(b).

The EFT switch is further implemented in terms of two components, namely, the financial component and the reversal component. A simplified behavioral model of the financial component is given in Figure 4(c). Comparing the two languages of $\bar{s}$ and $\bar{e}$, t action (representing time-out) is considered as an internal

interface between $\bar{e}$ and a to-be-developed implementation of the reversal component. Observe that for every sequence σ in $\{p\_rq(\delta_e|rev\_rq)^*, p\_rq(\delta_e|rev\_rq)^* rev\_rq(\delta|\delta_e)^*, p\_rq\ p\_rs(\delta_e|rev\_rq)^*(\delta|\delta_e)^*, (\delta_e|rev\_rq)^*, (\delta_e|rev\_rq)^* rev\_rq(rev\_rq|p\_rq)^*(\delta|\delta_e)^*\}$, it holds that $\mathsf{out}(\mathbf{hide}[t]\,\mathbf{in}\,\bar{e}\ \ \mathsf{after}\ \sigma_{\downarrow_{L_e}}) \subseteq \mathsf{out}(\bar{s}\ \mathsf{after}\ \sigma)$; thus, the behavior of $\bar{e}$ is included in $\bar{s}$. We next instigate investigate decomposability of $\bar{s}$ with $\bar{e}$, by constructing the quotient $\bar{s}_{/\bar{e}}$. Note that t is the only shared action which is an input action from the view point of $\bar{s}_{/\bar{e}}$. The resulting quotient automaton, obtained by applying Definition 15 to $\bar{s}$ and $\bar{e}$ is depicted in Figure 4(d). We illustrate some steps in its derivation. The initial state of the quotient automaton is defined as the $\{(\bar{s},\bar{e})\}$. Below, we illustrate which of the rules of Definition 15 are possible from this initial state; doing so repeatedly for all reached states will ultimately produce the reachable states of the quotient automaton.

1. We check the possibility of adding input transitions to the initial state, *i.e.* $q_0 = \{(\bar{s},\bar{e})\}$. Following q_0 $\mathsf{executes}$ $t = \{(s_1,e_2)\}$ and deduction rule I_1 in Definition 15, the transition $q_0 \xrightarrow{t} q_1$ is added to the transition relation of $\bar{s}_{/\bar{e}}$ where $q_1 = \{(s_1,e_2)\}$ (state 1 in Figure 4(d)).
2. We check the possibility of adding output transitions to $q_0 = \{(\bar{s},\bar{e})\}$. We observe that $rev\_rq \in \mathsf{out}(\bar{s}\ \mathsf{after}\ \sigma)$ for every $\sigma \in \{\varepsilon, p\_rq, p\_rq\ \ p\_rs\}$. Regarding deduction rule U_2, the transition $q_0 \xrightarrow{rev\_rq} q_2$ is added to the transition relation of $\bar{s}_{/\bar{e}}$ where $q_2 = \{(s_5,\bar{e}),(s_2,e_1),(s_2,e_3)\}$ (state 2 in Figure 4(d)).
3. Following deduction rule δ_1 and $\delta \notin \mathsf{out}(\bar{s}\ \mathsf{after}\ \varepsilon)$, δ-labeled transition is not added to q_0.

The constructed quotient automaton $\bar{s}_{/\bar{e}}$ is valid: it is both a valid suspension automaton and strongly non-blocking. As a result, $\bar{s}$ is decomposable with respect to $\bar{e}$ and $\bar{s}_{/\bar{e}}$ is a witness to that.

5 Strong Decomposibility

It is a natural question whether the quotient automaton that we defined in the previous section, along with the sufficient conditions for decomposability of a specification provide sufficient conditions for strong decomposability. The proof of Theorem 1 gives some clues to the contrary. A main problem is in the notion of quiescence, and, in particular in the notion of *relative* quiescence, which is unobservable in the standard **ioco** theory. More specifically, the platform $\bar{e}$ may mask the (unwanted) lack of outputs of the quotient automaton.

A natural solution to this is to consider a subclass of implementations called *internal choice* IOTSs, studied in [8, 15]: such implementations *only* accept inputs when reaching a quiescent state. The proposition below states that strong decomposability can be achieved under these conditions.

Theorem 2 *Let $\bar{s} \in \mathsf{IOLTS}(I_s,U_s)$ be a specification and let $\bar{e} \in \mathsf{IOTS}(I_e,U_e)$ be an environment. If $\bar{s}$ is decomposable and $\bar{e}$ is an internal choice IOTS then $\bar{s}$ is* strongly decomposable *and $\bar{s}_{/\bar{e}}$ is a witness to this.*

As a result of the above theorem, testing whether the composition of a component $\bar{c}$ and a platform $\bar{e}$ conforms to specification $\bar{s}$ reduces to testing for the conformance of $\bar{c}$ to $\bar{s}_{/\bar{e}}$. This can be done using the standard **ioco** testing theory [13].

A problem may arise when trying this approach in practice. Namely, the amount of time and memory needed for derivation of the **ioco** test suit increases exponentially in the number of transitions in the specification due to the nondeterministic nature of the test-case generation algorithm. We avoid these complexities by presenting an on-the-fly testing algorithm inspired by [4]. Algorithm 1 describes the

on-the-fly testing algorithm in which sound test cases are generated without constructing the quotient automaton upfront. We partially explored the quotient automaton during test execution. We use the extended version of executes operator in Algorithm 1 which is defined on ordinary IOLTSs; the underlying IOLTSs of suspension automata is used to avoid the complexity of constructing suspension automata, *i.e.* executes $: \mathbb{P}(\mathbb{P}(S_s) \times \mathbb{P}(S_e)) \times L_\delta \times \mathbb{P}(\mathbb{P}(S_s) \times \mathbb{P}(S_e))$.

Algorithm 1 *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification and let $\bar{e} \in \mathsf{IOTS}(I_e, U_e)$ be an environment. Let $\bar{c} \in \mathsf{IOTS}(L_I, L_U)$ be an implementation tested against $\bar{s}$ with respect to $\bar{e}$ by application of the following rules, initializing S with $(\{(\bar{s}\ \mathsf{after}\ \varepsilon), (\bar{e}\ \mathsf{after}\ \varepsilon)\})$ and verdict V with* None*:*
while ($V \notin \{\text{Fail}, \text{Pass}\}$)
{ *apply one of the following case:*

1. *(*provide an input*) Select an $a \in \{a \in L_I \mid S\ \mathsf{executes}\ a \neq \emptyset\}$, then $S = S\ \mathsf{executes}\ a$ and provide $\bar{c}$ with a*
2. *(*accept quiescence*) If no output is generated by $\bar{c}$ (quiescence situation) and $\forall (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) :\ \delta \in \mathsf{out}(s\ \mathsf{after}\ \sigma)$, then $S = S\ \mathsf{executes}\ \delta$*
3. *(*fail on quiescence*) If no output is generated by $\bar{c}$ (quiescence situation) and $(\exists (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) :\ \delta \notin \mathsf{out}(s\ \mathsf{after}\ \sigma))$, then $V = \text{Fail}$*
4. *(*accept a shared output*) If $x \in U_v$ is produced by $\bar{c}$ and $S\ \mathsf{executes}\ x \neq \emptyset$, then $S = S\ \mathsf{executes}\ x$*
5. *(*fail on a shared output*) If $x \in U_v$ is produced by $\bar{c}$ and $S\ \mathsf{executes}\ x = \emptyset$, then $V = \text{Fail}$*
6. *(*accept an output*) If $x \in U \setminus U_v$ is produced by $\bar{c}$ and $\forall (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) \cap (L_\delta^* \setminus L_\delta^* \delta) :\ x \in \mathsf{out}(s\ \mathsf{after}\ \sigma)$, then $S = S\ \mathsf{executes}\ x$*
7. *(*fail on an output*) If $x \in U \setminus U_v$ is produced by $\bar{c}$ and $\exists (s,e) \in S, \sigma \in \mathsf{Straces}(s) \cap \mathsf{Straces}(e) \cap (L_\delta^* \setminus L_\delta^* \delta) :\ x \notin \mathsf{out}(s\ \mathsf{after}\ \sigma)$, then $V = \text{Fail}$*
8. *(*nondeterministically terminate*) $V = \text{Pass}$ }*

Termination of the above algorithm with $V = \mathit{Fail}$ implies that the composition of the implementation under test with $\bar{e}$ does not conform to $\bar{s}$.

Theorem 3 *Let $\bar{s} \in \mathsf{IOLTS}(I_s, U_s)$ be a specification and let $\bar{e} \in \mathsf{IOLTS}(I_e, U_e)$ be an internal choice IOTS environment whose behavior is included in $\bar{s}$. Let V be the verdict upon termination of Algorithm 1 when executed on an implementation $\bar{c}$. If $\mathbf{hide}[L_v]\,\mathbf{in}\,\bar{c} \| \bar{e}$ **ioco** $\bar{s}$ then $V = \text{Pass}$.*

6 Conclusions

We investigated the property of *decomposability* of a specification in the setting of Tretmans' **ioco** theory for formal conformance testing [12]. Decomposability allows for determining whether a specification can be met by some implementation running on a given platform. Based on a new specification, to which we refer to as the *quotient*, and which we derived from the given one by factoring out the effects of the platform, we identified three conditions (two on the quotient and one on the platform) that together guarantee the decomposability of the original specification.

Any component that correctly implements the quotient is guaranteed to work correctly on the given platform. However, failing implementations provide no information on the correctness of the cooperation between the component and the platform. We therefore studied *strong decomposability*, which

further strengthens the decomposability problem to ensure that only those components that correctly implement the quotient are guaranteed to work correctly on the given platform, meeting the overall specification. This ensures that testing a component against the quotient provides all information needed to judge whether it will work correctly on the platform and meet the overall specification's requirements. However, the complexity of computing the quotient is an exponential problem. We propose an on-the-fly test case derivation algorithm which does not compute the quotient explicitly. Components that fail such a test case provably fail to work on the platform, meeting the overall specification, too.

Checking the inclusion relation of a platform may be expensive in practice. As for future work, we would like to merge the two steps of checking the correctness of the platform and driving the quotient and investigate whether the constraints on the platform can be relaxed by ensuring that the derived quotient masks some of the unwanted behavior of the platform.

References

[1] H. R. Asaadi, R. Khosravi, M. R. Mousavi & N. Noroozi (2011): *Towards Model-Based Testing of Electronic Funds Transfer Systems*. In: *FSEN*, *LNCS* 7141, pp. 253–267. Available at `http://dx.doi.org/10.1007/978-3-642-29320-7_17`.

[2] S. Berezin, S. Campos & E.M. Clarke (1998): *Compositional Reasoning in Model Checking*. In: *Compositionality: The Significant Difference*, *LNCS* 1536, Springer, pp. 81–102. Available at `http://dx.doi.org/10.1007/3-540-49213-5_4`.

[3] M. van der Bijl, A. Rensink & J. Tretmans (2003): *Compositional Testing with ioco*. In: *FATES*, *LNCS* 2931, Springer, pp. 86–100. Available at `http://dx.doi.org/10.1007/978-3-540-24617-6_7`.

[4] R.G. de Vries & J. Tretmans (2000): *On-the-fly Conformance Testing using SPIN*. *STTT* 2(4), pp. 382–393. Available at `http://dx.doi.org/10.1007/s100090050044`.

[5] L. Frantzen & J. Tretmans (2006): *Model-Based Testing of Environmental Conformance of Components*. In: *FMCO*, *LNCS* 4709, Springer, pp. 1–25. Available at `http://dx.doi.org/10.1007/978-3-540-74792-5_1`.

[6] D. Giannakopoulou, C. S. Pasareanu & H. Barringer (2005): *Component Verification with Automatically Generated Assumptions*. *Autom. Softw. Eng.* 12(3), pp. 297–320. Available at `http://dx.doi.org/10.1007/s10515-005-2641-y`.

[7] O. Kupferman & M. Vardi (1998): *Modular model checking*. In: *Compositionality: The Significant Difference*, *LNCS* 1536, Springer, pp. 81–102. Available at `http://dx.doi.org/10.1007/3-540-49213-5_4`.

[8] N. Noroozi, R. Khosravi, M.R. Mousavi & T. A. C. Willemse (2011): *Synchronizing Asynchronous Conformance Testing*. In: *SEFM*, *LNCS* 7041, Springer, pp. 334–349. Available at `http://dx.doi.org/10.1007/978-3-642-24690-6_23`.

[9] N. Noroozi, M.R. Mousavi & T.A.C. Willemse (2013): *Decomposability in Formal Conformance Testing*. Technical Report CSR-13-02, TU/Eindhoven.

[10] C. S. Pasareanu, M. B. Dwyer & M. Huth (1999): *Assume-Guarantee Model Checking of Software: A Comparative Case Study*. In: *Theoretical and Practical Aspects of SPIN Model Checking*, *LNCS* 1680, Springer, pp. 168–183. Available at `http://dx.doi.org/10.1007/3-540-48234-2_14`.

[11] A. Simão & A. Petrenko (2011): *Generating asynchronous test cases from test purposes*. *Information & Software Technology* 53(11), pp. 1252–1262. Available at `http://dx.doi.org/10.1016/j.infsof.2011.06.006`.

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

[13] J. Tretmans (2008): *Model Based Testing with Labelled Transition Systems*. In: *Formal Methods and Testing*, *LNCS* 4949, Springer, pp. 1–38. Available at http://dx.doi.org/10.1007/978-3-540-78917-8_1.

[14] T. Villa, N. Yevtushenko, R.K. Brayton, A. Mishchenko, A. Petrenko & A. Sangiovanni-Vincentelli (2012): *The Unknown Component Problem, Theory and Applications*. Springer, doi:10.1007/978-0-387-68759-9.

[15] M. Weiglhofer & F. Wotawa (2009): *Asynchronous Input-Output Conformance Testing*. In: *COMPSAC*, IEEE Computer Society, pp. 154–159. Available at http://dx.doi.org/10.1109/COMPSAC.2009.194.

[16] T. A. C. Willemse (2006): *Heuristics for ioco -Based Test-Based Modelling*. In: *FMICS/PDMC*, *LNCS* 4346, Springer, pp. 132–147. Available at http://dx.doi.org/10.1007/978-3-540-70952-7_9.