Compositional Testing with IOCO

Machiel van der Bijl[1] *, Arend Rensink[1] and Jan Tretmans[2]

[1] Software Engineering, Department of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{vdbijl, rensink}@cs.utwente.nl
[2] Software Technology Research Group, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
tretmans@cs.kun.nl

Abstract. *Compositional testing* concerns the testing of systems that consist of communicating components which can also be tested in isolation. Examples are component based testing and interoperability testing. We show that, with certain restrictions, the **ioco**-test theory for conformance testing is suitable for compositional testing, in the sense that the integration of fully conformant components is guaranteed to be correct. As a consequence, there is no need to re-test the integrated system for conformance.
This result is also relevant for *testing in context*, since it implies that every failure of a system embedded in a test context can be reduced to a fault of the system itself.

1 Introduction

In this paper we study formal testing based on the **ioco**-test theory. This theory works on labeled transition systems (LTS) [1,2]. The name **ioco**, which stands for *input/output conformance*, refers to the implementation relation (i.e., notion of correctness) on which the theory and the test generation algorithm have been built. A number of tools are based on the **ioco** theory, among which there are TGV [3], TESTGEN [4] and TorX [5].

Two open issues in testing theory in general, and the **ioco**-theory in particular, are *compositional testing* and *testing in context*. For instance, for the testing theory based on Finite-State-Machines (FSM) this issue has been studied in [6].

Compositional testing considers the testing of communicating components that together form a larger system. An example is component based testing, i.e., integration testing of components that have already been tested separately. An example from the telecom sector is interoperability testing, i.e., testing if systems from different manufacturers, that should comply with a certain standard, work together; for example GSM mobile phones. The question is what can be concluded from the individual tests of the separate components, and what should be (re)tested on the integration or system level. With the current theory it is unclear what the relation between the correctness of the components and the integrated system is.

* This research was supported by Ordina Finance and by the dutch research programme PROGRESS under project: TES5417: Atomyste – ATOm splitting in eMbedded sYStems TEsting.

Another scenario, with similar characteristics, is *testing in context*. This refers to the situation that a tester can only access the implementation under test through a *test context* [7,8,9]. The test context interfaces between the implementation under test and the tester. As a consequence the tester can only indirectly observe and control the IUT via the test context. This makes testing weaker, in the sense that there are fewer possibilities for observation and control of the IUT. With testing in context, the question is whether faults in the IUT can be detected by testing the composition of IUT and test context, and whether a failure of this composition always indicates a fault of the IUT. This question is the converse of compositional testing: when testing in context we wish to detect errors in the IUT — a component — by testing it in composition with the test context, whereas in compositional testing we wish to infer correctness of the integrated system from conformance of the individual components.

This paper studies the above mentioned compositionality properties of **ioco** for two operations on labeled transition systems: parallel composition and hiding. If **ioco** has this compositionality property for these operations, it follows that correctness of the parts (the components) implies correctness of the whole (the integrated system), or that a fault in the whole (IUT and test context) implies a fault in the component (IUT). This compositionality property is formally called a pre-congruence.

We show that **ioco** is a pre-congruence for parallel composition and hiding in the absence of underspecification of input actions. One way to satisfy this condition is to only allow specifications which are *input enabled*. Another way is to make the underspecification explicit by *completion*. We show that, in particular, *demonic completion* is suitable for this purpose. As a final result we show how to use the original (uncompleted) specifications and still satisfy the pre-congruence property. This leads to a new implementation relation, baptized $\mathbf{ioco}_U$ which is slightly weaker than **ioco**.

This paper has two main results. First we show a way to handle underspecification of input actions when testing communicating components with the **ioco** theory. This idea is new for LTS testing. It is inspired by [10] and similar work done in FSM testing [11]. Second we establish a formal relation between the components and the integrated system. As far as we know this result is new for both LTS testing and FSM testing.

Overview. The next section recalls some basic concepts and definitions about transition systems and **ioco**. Section 3 sets the scene and formalizes the problems of compositional testing and testing in context. Section 4 studies the pre-congruence properties of **ioco** for parallel composition and hiding. Section 5 discusses underspecification, and approaches to complete specifications with implicit underspecification. Section 6 concludes with some final remarks and an assessment of the results. For a full version of this paper with all the proofs, we refer to [12].

2 Formal preliminaries

This section recalls the aspects of the theory behind **ioco** that are used in this paper; see [1] for a more detailed exposition.

Labeled Transition Systems. A labeled transition system (LTS) description is defined in terms of states and labeled transitions between states, where the labels indicate what happens during the transition. Labels are taken from a global set $\mathbf{L}$. We use a special label $\tau \notin \mathbf{L}$ to denote an internal action. For arbitrary $L \subseteq \mathbf{L}$, we use L_τ as a shorthand for $L \cup \{\tau\}$. We deviate from the standard definition of labeled transition systems in that we assume the label set of an LTS to be partitioned in an input and an output set.

Definition 1. *A* labeled transition system *is a 5-tuple* $\langle Q, I, U, T, q_0 \rangle$ *where* Q *is a non-empty countable set of* states*;* $I \subseteq \mathbf{L}$ *is the countable set of* input labels*;* $U \subseteq \mathbf{L}$ *is the countable set of* output labels*, which is disjoint from* I*;* $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ *is a set of triples, the* transition relation*;* $q_0 \in Q$ *is the* initial state*.*

We use L as shorthand for the entire label set ($L = I \cup U$); furthermore, we use Q_p, I_p etc. to denote the components of an LTS p. We commonly write $q \xrightarrow{\lambda} q'$ for $(q, \lambda, q') \in T$. Since the distinction between inputs and outputs is important, we sometimes use a question mark before a label to denote input and an exclamation mark to denote output. We denote the class of all labeled transition systems over I and U by $\mathcal{LTS}(I, U)$. We represent a labeled transition system in the standard way, by a directed, edge-labeled graph where nodes represent states and edges represent transitions.

A state that cannot do an internal action is called *stable*. A state that cannot do an output or internal action is called *quiescent*. We use the symbol δ ($\notin \mathbf{L}_\tau$) to represent quiescence: that is, $p \xrightarrow{\delta} p$ stands for the absence of any transition $p \xrightarrow{\lambda} p'$ with $\lambda \in U_\tau$. For an arbitrary $L \subseteq \mathbf{L}_\tau$, we use L_δ as a shorthand for $L \cup \{\delta\}$.

An LTS is called *strongly responsive* if it always eventually enters a quiescent state; in other words, if it does not have any infinite U_τ-labeled paths. For technical reasons we restrict $\mathcal{LTS}(I, U)$ to strongly responsive transition systems. Systems that are not strongly responsive may show live-locks (or develop live-locks by hiding actions). So one can argue that it is a favorable property if a specification is strongly responsive. However, from a practical perspective it would be nice if the constraint can be lessened. This is probably possible, but needs further research.

A *trace* is a finite sequence of observable actions. The set of all traces over L ($\subseteq \mathbf{L}$) is denoted by L^*, ranged over by σ, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. We use the standard notation with single and double arrows for traces: $q \xrightarrow{a_1 \cdots a_n} q$ denotes $q \xrightarrow{a_1} \cdots \xrightarrow{a_n} q'$, $q \xRightarrow{\epsilon} q'$ denotes $q \xrightarrow{\tau \cdots \tau} q'$ and $q \xRightarrow{a_1 \cdot \ldots \cdot a_n} q$ denotes $q \xRightarrow{\epsilon} \xrightarrow{a_1} \xRightarrow{\epsilon} \cdots \xrightarrow{a_n} \xRightarrow{\epsilon} q'$ (where $a_i \in \mathbf{L}_{\tau\delta}$).

We will not always distinguish between a labeled transition system and its initial state. We will identify the process $p = \langle Q, I, U, T, q_0 \rangle$ with its initial state q_0, and we write, for example, $p \xRightarrow{\sigma} q_1$ instead of $q_0 \xRightarrow{\sigma} q_1$.

Input-output transition systems. An *input-output transition system* (IOTS) is a labeled transition system that is completely specified for input actions. The class of input-output transition systems with input actions in I and output actions in U is denoted by $\mathcal{IOTS}(I, U)$ ($\subseteq \mathcal{LTS}(I, U)$). Notice that we do not require IOTS's to be strongly responsive.

Definition 2. *An* input-output transition system $p = \langle Q, I, U, T, q_0 \rangle$ *is a labeled transition system for which all inputs are enabled in all states:* $\forall q \in Q, a \in I : q \xRightarrow{a}$

Composition of labeled transition systems. The integration of components can be modeled algebraically by putting the components in parallel while synchronizing their common actions, possibly with internalizing (hiding) the synchronized actions. In process algebra, the synchronization and internalization are typically regarded as two separate operations. The synchronization of the processes p and q is denoted by $p \,\|\, q$. The internalization of a label set V in process p, or *hiding* V in p as it is commonly called, is denoted by $\mathbf{hide}\ V\ \mathbf{in}\ p$. Below we give the formal definition.

Definition 3. *For $i = 1, 2$ let $p_i = \langle Q_i, I_i, U_i, T_i, p_i \rangle$ be a transition system.*
- *If $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ then $p_1 \,\|\, p_2 =_{\mathrm{def}} \langle Q, I, U, T, p_1 \,\|\, p_2 \rangle$ where*
  - $Q = \{q_1 \,\|\, q_2 \mid q_1 \in Q_1, q_2 \in Q_2\}$*;*
  - $I = (I_1 \setminus U_2) \cup (I_2 \setminus U_1)$*;*
  - $U = U_1 \cup U_2$.
  - *T is the minimal set satisfying the following inference rules ($\mu \in L_\tau$):*

$$\begin{array}{ll} q_1 \xrightarrow{\mu} q_1',\ \mu \notin L_2 & \vdash q_1 \,\|\, q_2 \xrightarrow{\mu} q_1' \,\|\, q_2 \\ q_2 \xrightarrow{\mu} q_2',\ \mu \notin L_1 & \vdash q_1 \,\|\, q_2 \xrightarrow{\mu} q_1 \,\|\, q_2' \\ q_1 \xrightarrow{\mu} q_1',\ q_2 \xrightarrow{\mu} q_2',\ \mu \neq \tau & \vdash q_1 \,\|\, q_2 \xrightarrow{\mu} q_1' \,\|\, q_2' \end{array}$$

- *If $V \subseteq U_1$, then $\mathbf{hide}\ V\ \mathbf{in}\ p_1 =_{\mathrm{def}} \langle Q, I_1, U_1 \setminus V, T, \mathbf{hide}\ V\ \mathbf{in}\ p_1 \rangle$ where*
  - $Q = \{\mathbf{hide}\ V\ \mathbf{in}\ q_1 \mid q_1 \in Q_1\}$*;*
  - *T is the minimal set satisfying the following inference rules ($\mu \in L_\tau$):*

$$\begin{array}{l} q_1 \xrightarrow{\mu} q_1',\ \mu \notin V \vdash \mathbf{hide}\ V\ \mathbf{in}\ q_1 \xrightarrow{\mu} \mathbf{hide}\ V\ \mathbf{in}\ q_1' \\ q_1 \xrightarrow{\mu} q_1',\ \mu \in V \vdash \mathbf{hide}\ V\ \mathbf{in}\ q_1 \xrightarrow{\tau} \mathbf{hide}\ V\ \mathbf{in}\ q_1' \end{array}$$

Note that these constructions are only *partial*: there are constraints on the input and output sets. Moreover, parallel composition may give rise to an LTS that is not strongly responsive, even if the components are. For the time being, we do not try to analyze this but implicitly restrict ourselves to cases where the parallel composition *is* strongly responsive (thus, this is another source of partiality of the construction).

In this paper we restrict ourselves to binary parallel composition. N-ary parallel composition may be an interesting extension. One may wonder however what this means in our input output setting, since an output action is uniquely identified by its sender. From this perspective only the synchronization of many receivers to one sender (broadcast) seems an interesting extension.

Proposition 1. *Let $p, q \in \mathcal{LTS}(I_i, U_i)$ for $i = p, q$, with $I_p \cap I_q = U_p \cap U_q = \emptyset$, and let $V \subseteq U_p$.*
1. *If $p \,\|\, q$ is strongly responsive then $p \,\|\, q \in \mathcal{LTS}((I_p \setminus U_q) \cup (I_q \setminus U_p), U_p \cup U_q)$; moreover, $p \,\|\, q \in \mathcal{IOTS}$ if $p, q \in \mathcal{IOTS}$.*
2. *$\mathbf{hide}\ V\ \mathbf{in}\ p \in \mathcal{LTS}(I_p, U_p \setminus V)$; moreover, $\mathbf{hide}\ V\ \mathbf{in}\ p \in \mathcal{IOTS}$ if $p \in \mathcal{IOTS}$.*

Conformance. The testing scenario on which $\mathbf{ioco}$ is based assumes that two things are given: 1) An LTS constituting a specification of required behavior. And 2) an implementation under test. We treat the IUT as a black box. In order to reason about it we assume it can be modeled as an IOTS (an IUT is an object in the real world) . This assumption is referred to as the test hypothesis [7]. We want to stress that we do not need to *have* this model when testing the IUT. We only *assume* that the implementation *behaves* as an IOTS.

Given a specification s and an (assumed) model of the IUT i, the relation i $\mathbf{ioco}$ s expresses that i conforms to s. Whether this holds is decided on the basis of the *suspension traces* of s: it must be the case that, after any such trace σ, every output action (and also quiescence) that i is capable of should be allowed according to s. This is formalized by defining p $\mathbf{after}$ σ (the set of states that can be reached in p after the suspension trace σ), $out(p)$ (the set of output and δ-actions of p) and $Straces(p)$ (the suspension traces of p).

Definition 4. *Let $p, s \in \mathcal{LTS}(I, U)$, let $i \in \mathcal{IOTS}(I, U)$, let $P \subseteq Q_p$ be a set of states in p and let $\sigma \in \mathbf{L}_\delta^*$.*

1. $p\,\mathbf{after}\ \sigma =_{\mathrm{def}} \{\ p' \mid p \stackrel{\sigma}{\Longrightarrow} p'\ \}$
2. $out(p) =_{\mathrm{def}} \{x \in U\ \mid p \stackrel{x}{\longrightarrow}\} \cup \{\delta \mid p \stackrel{\delta}{\longrightarrow}\}$
3. $out(P) =_{\mathrm{def}} \bigcup\ \{\ out(p) \mid p \in P\ \}$
4. $Straces(p) =_{\mathrm{def}} \{\sigma \in L_\delta^* \mid p \stackrel{\sigma}{\Longrightarrow}\}$

The following defines the implementation relation $\mathbf{ioco}$, modulo a function $\mathcal{F}$ that generates a set of test-traces from a specification. In this definition 2^X denotes the powerset of X, for an arbitrary set X.

Definition 5. *Given a function $\mathcal{F} : \mathcal{LTS}(I, U) \to 2^{L_\delta^*}$, we define the implementation relation $\mathbf{ioco}_\mathcal{F} \subseteq \mathcal{IOTS}(I, U) \times \mathcal{LTS}(I, U)$ as follows:*

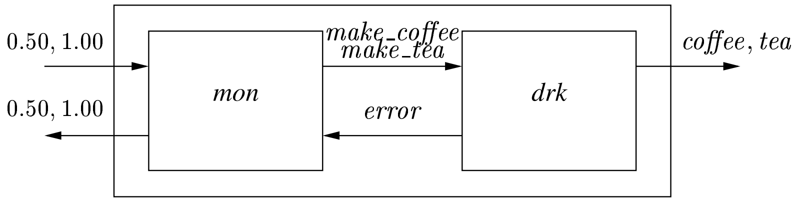
$$i\ \mathbf{ioco}_\mathcal{F}\ s \iff \forall \sigma \in \mathcal{F}(s) : out(i\ \mathbf{after}\ \sigma) \subseteq out(s\ \mathbf{after}\ \sigma)$$

So i $\mathbf{ioco}_{Straces}$ s means $\forall \sigma \in Straces(s) : out(i\ \mathbf{after}\ \sigma) \subseteq out(s\ \mathbf{after}\ \sigma)$. We use $\mathbf{ioco}$ as an abbreviation for $\mathbf{ioco}_{Straces}$. For more detailed information about $\mathbf{ioco}$ we refer to [1].

3 Approach

In this section we want to clarify compositional testing with the formal framework presented in the previous section. The consequences for testing in context will be discussed in the final section.

We study systems that consist of communicating components. These components can be tested individually and while working together (in the case of testing in context the components are the IUT and its test context). The behavior of such a system is described by the parallel composition of the individual transition systems. Output actions of one component that are in the input label set of another component are *synchronized*, resulting in a single, internal transition of the overall system. Actions of a component that are not in the label set of another component are not synchronized, resulting in a single observable transition of the overall system. This gives rise to the scenario depicted in Figure 1. The figure will be explained in the next example.



$$s_{cof} = \textbf{hide}\ \{make\_coffee, make\_tea, error\}\ \textbf{in}\ s_{mon} \,\|\, s_{drk}$$
$$i_{cof} = \textbf{hide}\ \{make\_coffee, make\_tea, error\}\ \textbf{in}\ i_{mon} \,\|\, i_{drk}$$

Fig. 1. Architecture of coffee machine in components

3.1 Example

To illustrate compositional testing, we use two components of a coffee machine: a "money component" (*mon*) that handles the inserted coins and a "drink component" (*drk*) that takes care of preparing and pouring the drinks, see Figure 1.

The money component accepts coins of €1 and of €0.50 as input from the environment. After insertion of a €0.50 coin (respectively €1 coin), the money component orders the drink component to make tea (respectively coffee).

The drink component interfaces with the money component and the environment. If the money component orders it to make tea (respectively coffee) it outputs tea (respectively coffee) to the environment. If anything goes wrong in the drink making process, the component gives an error signal.

The coffee machine is the parallel composition of the money component and the drink component, in which the "make coffee" command, the "make tea" command and the "error" signal are hidden. One can think of the parallel composition as establishing the connection between the money component and the drink component, whereas hiding means that the communication between the components is not observable anymore; only communication with the environment can be observed.

Models. In Figure 2 we show the behavioral specification of the money component s_{mon} and the drink component s_{drk} as LTS's. Note that the money component is underspecified for the *error* input label and that the drink component cannot recover from an error state, and while in the error state it cannot produce tea or coffee. Figure 3 shows implementation models of the money component, i_{mon}, and the drink component, i_{drk}. We have used transitions labeled with '?' as an abbreviation for all the non-specified input actions from the alphabet of the component. The money component has input label set, $I_{mon} = \{\mathit{0.50}, \mathit{1.00}, error\}$, output label set $U_{mon} = \{make\_coffee, make\_tea, \mathit{0.50}, \mathit{1.00}\}$; specification $s_{mon} \in \mathcal{LTS}(I_{mon}, U_{mon})$, and implementation $i_{mon} \in \mathcal{IOTS}(I_{mon}, U_{mon})$. $I_{drk} = \{make\_coffee, make\_tea\}$ and $U_{drk} = \{coffee, tea, error\}$ are the input and output label set respectiveley and $s_{drk} \in \mathcal{LTS}(I_{drk}, U_{drk}), i_{drk} \in \mathcal{IOTS}(I_{drk}, U_{drk})$.

In the implementations of the components we choose to improve upon the specification, by adding functionality. This is possible since $\textbf{ioco}$ allows partial specifications.

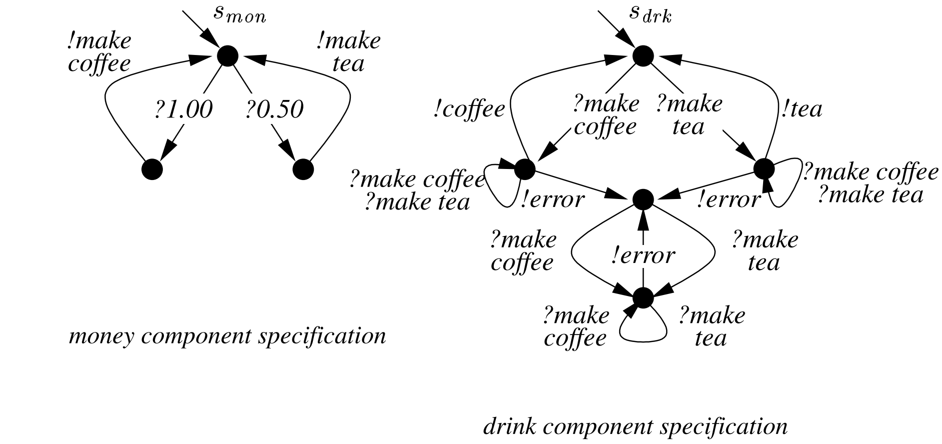


Fig. 2. Specification of money and drink components as LTS's

Implementers are free to make use of the underspecification. The extra functionality of i_{mon} compared to its specification s_{mon} is that it can handle error signals: it reacts by returning €1.00. i_{drk} is also changed with respect to its specification s_{drk}: making tea never produces an error signal. Since implementations are input enabled, we have chosen that all non specified inputs are ignored, i.e., the system remains in the same state.

We have i_{mon} **ioco** s_{mon} and i_{drk} **ioco** s_{drk}. The question now is whether the integrated implementation, as given by i_{cof} in Figure 1, is also **ioco** correct with respect to the integrated specification s_{cof}. We discuss this in section 4, to illustrate the compositionality properties discussed there.

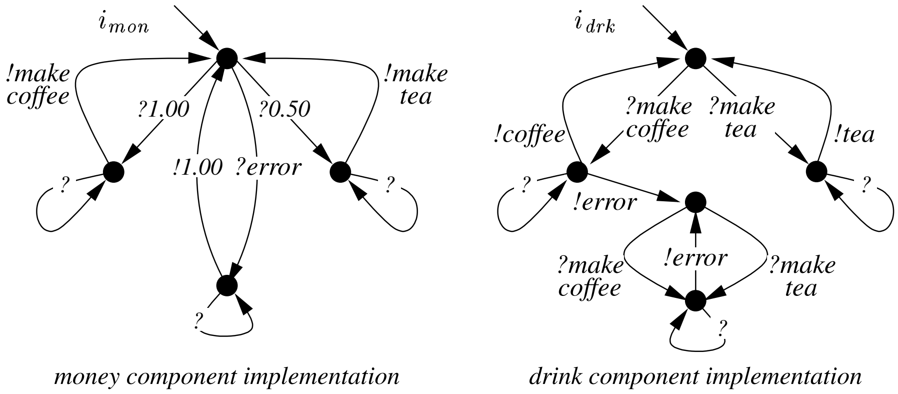


Fig. 3. Implementation of the money and drink components as IOTS's

3.2 Compositional testing

We now paraphrase the question of compositional testing, discussed in the introduction, as follows: "Given that the components p and q have been tested to be $\mathbf{ioco}$-correct (according to their respective specifications), may we conclude that their integration is also $\mathbf{ioco}$-correct (according to the integrated specification)?" If the component specifications are LTS's, the component implementations are modeled by IOTS's, and their integration by parallel composition followed by hiding, this boils down to the following questions in our formal framework (where $i_k \in \mathcal{IOTS}(I_k, U_k)$ and $s_k \in \mathcal{LTS}(I_k, U_k)$ for $k = 1, 2$, with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$):

Q1: Given i_k $\mathbf{ioco}$ s_k for $k = 1, 2$, is it the case that $i_1 \parallel i_2$ $\mathbf{ioco}$ $s_1 \parallel s_2$?
Q2: Given i_1 $\mathbf{ioco}$ s_1, is it the case that $(\mathbf{hide}\ V\ \mathbf{in}\ i_1)$ $\mathbf{ioco}$ $(\mathbf{hide}\ V\ \mathbf{in}\ s_1)$ for arbitrary $V \subseteq U_1$?

If the answer to both questions is "yes", then we may conclude that $\mathbf{ioco}$ is suitable for compositional testing as stated in the following corollary.

Conjecture 1. If $i_k \in \mathcal{IOTS}(I_k, U_k)$ and $s_k \in \mathcal{LTS}(I_k, U_k)$ for $k = 1, 2$ with $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$ and $V = (I_1 \cap U_2) \cup (U_1 \cap I_2)$, then

$$i_1\ \mathbf{ioco}\ s_1 \wedge i_2\ \mathbf{ioco}\ s_2 \Rightarrow (\mathbf{hide}\ V\ \mathbf{in}\ i_1 \parallel i_2)\ \mathbf{ioco}\ (\mathbf{hide}\ V\ \mathbf{in}\ s_1 \parallel s_2)\ .$$

We study the above pre-congruence questions in the next section. We will show that the answer to Q1 and Q2 in general is *no*. Instead, we can show that the answer to Q1 and Q2 is *yes* if s_1 and s_2 are completely specified.

4 Compositionality for synchronization and hiding

In this section we address the questions Q1 and Q2 formulated above (Section 3.2), using the coffee machine example to illustrate our results.

4.1 Synchronization

The property that we investigate for parallel composition is: *if* we have two correct component implementations according to $\mathbf{ioco}$, *then* the implementation remains correct after synchronizing the components. It turns out that in general this property does not hold, as we show in the following example.

Example 1. Regard the LTS's in figure 4. On the left hand side we show the specifications and on the right hand side the corresponding implementations. The models have the following label sets: $s_1 \in \mathcal{LTS}(\{x\}, \emptyset), i_1 \in \mathcal{IOTS}(\{x\}, \emptyset), s_2 \in \mathcal{LTS}(\emptyset, \{x\})$ and $i_2 \in \mathcal{IOTS}(\emptyset, \{x\})$. The suspension traces of s_1 are given by $\delta^* \cup \delta^*?x\delta^*$ and the suspension traces of s_2 are given by $\{\epsilon, !x\} \cup !x!x\delta^*$. We have i_1 $\mathbf{ioco}$ s_1 and i_2 $\mathbf{ioco}$ s_2.

After we take the parallel composition of the two specifications we get $s_1 \parallel s_2$, see figure 4 (the corresponding implementation is $i_1 \parallel i_2$). We now see the following: $out(i_1 \parallel i_2\,\mathbf{after}\,!x) = \{!x\} \not\subseteq out(s_1 \parallel s_2\,\mathbf{after}\,!x) = \{\delta\}$; this means that the parallel composition of the implementations is not $\mathbf{ioco}$-correct: $i_1 \parallel i_2$ $\not\mathbf{ioco}$ $s_1 \parallel s_2$.

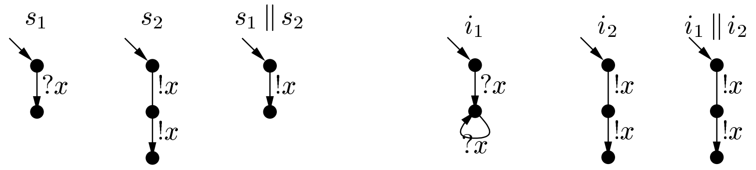


Fig. 4. Counter example to compositionality for parallel composition; see Example 1

Analysis shows that i_1 **ioco** s_1, because **ioco** allows underspecification of input actions. However, the semantics of the parallel composition operator does not take underspecification of input actions into account. Although s_2 can output a second x, it cannot do so in $s_1 \,||\, s_2$, because s_1 cannot input the second x.

It turns out that if we forbid implicit underspecification, i.e., if the specification explicitly prescribes for any possible input what the allowed responses are, then we do not have this problem. In fact in that case we have the desired compositionality property. This property is expressed in the following theorem. For a proof see [12].

Theorem 1. *Let* $s_1, i_1 \in \mathcal{IOTS}(I_1, U_1)$, $s_2, i_2 \in \mathcal{IOTS}(I_2, U_2)$, *with* $I_1 \cap I_2 = U_1 \cap U_2 = \emptyset$.

$$i_1 \;\mathbf{ioco}\; s_1 \wedge i_2 \;\mathbf{ioco}\; s_2 \Rightarrow i_1 \,||\, i_2 \;\mathbf{ioco}\; s_1 \,||\, s_2$$

Our running example (Section 3.1) shows the same problem illustrated in example 1. Although the implementations of the money component and the drink component are **ioco** correct with respect to their specifications, it turns out that the parallel composition of i_{mon} and i_{drk} is not:

$$\begin{array}{rcl} out(i_{mon} \,||\, i_{drk} \;\mathbf{after}\; ?1.00 \cdot !make\_coffee) & = & \{!coffee, !error\} \\ out(s_{mon} \,||\, s_{drk} \;\mathbf{after}\; ?1.00 \cdot !make\_coffee) & = & \{!coffee\} \end{array}$$

Note that the internal signals are still visible as output actions. To turn them into internal actions is the task of the *hiding* operator, discussed below.

4.2 Hiding

The property that we investigate for hiding is the following: *if* we have a correct implementation according to **ioco**, *then* the implementation remains correct after hiding (some of the) output actions. It turns out that, as for synchronization, in general this property does not hold.

Example 2. Consider the implementation i and specification s in Figure 5, both with input set $\{a\}$ and output set $\{x, y\}$. The suspension traces of s are $\{\epsilon\} \cup ?a\delta^* \cup !x\delta^*$. We see that i **ioco** s.

We get the specification $\mathbf{hide}\;\{x\}\;\mathbf{in}\;s$, and implementation $\mathbf{hide}\;\{x\}\;\mathbf{in}\;i$ after hiding the output action x. After the input a we get: $out(\mathbf{hide}\;\{x\}\;\mathbf{in}\;i\;\mathbf{after}\;a) = \{\delta, y\} \not\subseteq out(\mathbf{hide}\;\{x\}\;\mathbf{in}\;s\;\mathbf{after}\;a) = \{\delta\}$; in other words the **ioco** relation does not hold: $\mathbf{hide}\;\{x\}\;\mathbf{in}\;i \;\;\mathbf{i\!\!/\!oco}\;\; \mathbf{hide}\;\{x\}\;\mathbf{in}\;s$.

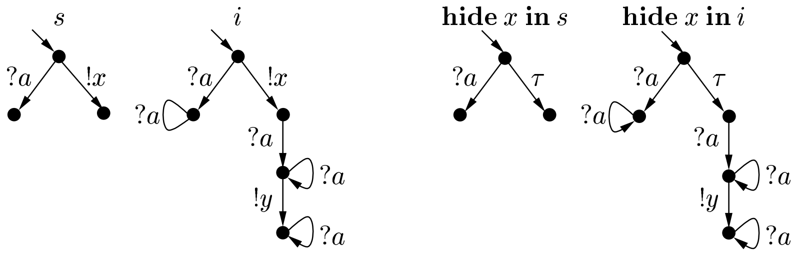


Fig. 5. Counter-example to compositionality for hiding; see Example 2

An analysis of the above example shows that s was underspecified, in the sense that it fails to prescribe how an implementation should behave after the trace $!x?a$. The proposed implementation i uses the implementation freedom by having an unspecified y-output after $!x?a$. However, if x becomes unobservable due to hiding, then the traces $!x?a$ and $?a$ collapse and become indistinguishable: in $\mathbf{hide}\ \{x\}\ \mathbf{in}\ s$ and $\mathbf{hide}\ \{x\}\ \mathbf{in}\ i$ they both masquerade as the trace $?a$. Now $\mathbf{hide}\ \{x\}\ \mathbf{in}\ s$ *appears* to specify that after $?a$, only quiescence (δ) is allowed; however, $\mathbf{hide}\ \{x\}\ \mathbf{in}\ i$ still has this unspecified y-output. In other words, hiding creates confusion about what part of the system is underspecified.

It follows that if we rule out underspecification, i.e., we limit ourselves to specifications that are IOTS's then this problem disappears. In fact, in that case we do have the desired congruence property. This is stated in the following theorem. For a proof see [12].

Theorem 2. *If* $i, s \in \mathcal{IOTS}(I, U)$ *with* $V \subseteq U$*, then:*

$$i \ \mathbf{ioco} \ s \ \Rightarrow \ (\mathbf{hide}\ V\ \mathbf{in}\ i) \ \mathbf{ioco} \ (\mathbf{hide}\ V\ \mathbf{in}\ s)$$

5 Demonic completion

We have shown in the previous section that $\mathbf{ioco}$ is a pre-congruence for parallel composition and hiding when restricted to $\mathcal{IOTS} \times \mathcal{IOTS}$. However, in the original theory [1] $\mathbf{ioco} \subseteq \mathcal{IOTS} \times \mathcal{LTS}$; the specifications are LTS's. The intuition behind this is that $\mathbf{ioco}$ allows underspecification of input actions. In this section we present a function that transforms LTS's into IOTS's in a way that complies with this notion of underspecification. We will show that this leads to a new implementation relation that is slightly weaker than $\mathbf{ioco}$.

Underspecification comes in two flavors: underspecification of input actions and underspecification of output actions. Underspecification of output actions is always explicit; in an LTS it is represented by a choice between several output actions. The intuition behind this is that we do not know or care which of the output actions is implemented, as long as at least one is. Underspecification of input actions is always implicit; it is represented by absence of the respective input action in the LTS. The intuition behind underspecification of input actions is that after an unspecified input action we do not know or care what the behavior of the specified system is. This means that in an underspecified state — i.e., a state reached after an unspecified input action — every action from the label set is correct, including quiescence. Following [13] we call this kind of behavior *chaotic*.

In translating LTS's to IOTS's, we propose to model underspecification of input actions explicitly. Firstly, we model chaotic behavior through a state q_χ with the property: $\forall \lambda \in U : q_\chi \overset{\lambda}{\Longrightarrow} q_\chi$ and $\forall \lambda \in I : q_\chi \overset{\delta^* \cdot \lambda}{\Longrightarrow} q_\chi$ (where χ stands for chaos). Secondly, we add for every stable state q (of a given LTS) that is underspecified for an input a, a transition (q, a, q_χ). This turns the LTS into an IOTS. After [10] we call this procedure *demonic* completion — as opposed to *angelic* completion, where unspecified inputs are discarded (modeled by adding self-loop transitions). Note that demonic completion results in an IOTS that is not strongly convergent. However the constraint of strong convergence only holds for LTS's.

Definition 6.
$\Xi : \mathcal{LTS}(I,U) \to \mathcal{IOTS}(I,U)$ is defined by $\langle Q, I, U, T, q_0\rangle \mapsto \langle Q', I, U, T', q_0\rangle$, where

$$Q' = Q \cup \{q_\chi, q_\Omega, q_\Delta\}, \text{ where } q_\chi, q_\Omega, q_\Delta \notin Q$$
$$T' = T \cup \{(q, a, q_\chi) \mid q \in Q, a \in I, q \overset{a}{\not\longrightarrow}, q \overset{\tau}{\not\longrightarrow}\}$$
$$\cup \{(q_\chi, \tau, q_\Omega), (q_\chi, \tau, q_\Delta)\} \cup \{(q_\Omega, \lambda, q_\chi) \mid \lambda \in L\} \cup \{(q_\Delta, \lambda, q_\chi) \mid \lambda \in I\}$$

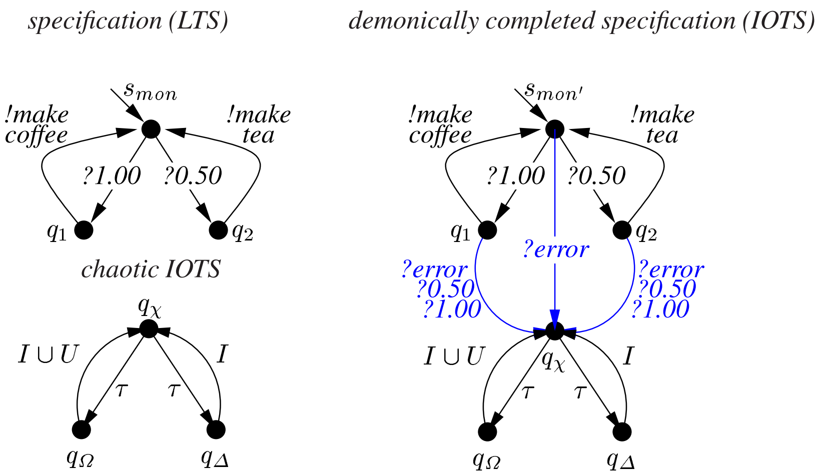


Fig. 6. Demonic completion of an LTS specification

Example 3. To illustrate the demonic completion of implicit underspecification, we use the money component of section 3.1. The LTS specification of the money component is given in the top left corner of Figure 6. The IOTS that models our chaos property is given in the bottom left corner. For every stable state of the specification that is underspecified for an input action, the function Ξ adds a transition with that input action to state q_χ. For example, every state is underspecified for input action $error$, so we add a transition from every state to q_χ for $error$. The states q_1 and q_2 are underspecified for 0.50 and 1.00, so we add transitions for these inputs from q_1 and q_2 to q_χ. The resulting demonically completed specification is given on the right hand side of Figure 6.

An important property of demonic completion is that it only adds transitions from *stable* states with underspecified inputs in the original LTS to q_χ. Moreover, it does not delete states or transitions. Furthermore, the chaotic IOTS acts as a kind of sink: once one of the added states (q_χ, q_Ω or q_Δ) has been reached, they will never be left anymore.

Proposition 2. *Let* $s \in \mathcal{LTS}(I, U)$. $\forall \sigma \in L_\delta^*, q' \in Q_s : s \stackrel{\sigma}{\Longrightarrow} q' \Leftrightarrow \Xi(s) \stackrel{\sigma}{\Longrightarrow} q'$

We use the notation "$\mathbf{ioco} \circ \Xi$" to denote that before applying $\mathbf{ioco}$, the LTS specification is transformed to an IOTS by Ξ; i.e., $i(\mathbf{ioco} \circ \Xi)s \Leftrightarrow i\ \mathbf{ioco}\ \Xi(s)$. This relation is slightly weaker than $\mathbf{ioco}$. This means that previously conformant implementations are still conformant, but it might be that previously non-allowed implementations are allowed with this new notion of conformance.

Theorem 3. $\mathbf{ioco} \subseteq \mathbf{ioco} \circ \Xi$

Note that the opposite is not true i.e., $i\ (\mathbf{ioco} \circ \Xi)\ s\ \not\Rightarrow i\ \mathbf{ioco}\ s$ (as the counterexamples of section 4 show). Furthermore this property is a consequence of our choice of the demonic completion function. Other forms of completion, such as angelic completion, result in variants of $\mathbf{ioco}$ which are incomparable to the original relation.

Testing. The testing scenario is now such that an integrated system can be tested by comparing the individual components to their *demonically completed* specifications. If the components conform, then the composition of implementations also conforms to the composition of the demonically completed specifications.

Corollary 1. *Let* $s_1, s_2 \in \mathcal{LTS}(I, U)$ *and* $i_1, i_2 \in \mathcal{IOTS}(I, U)$

$$i_1\ \mathbf{ioco}\ \Xi(s_1) \wedge i_2\ \mathbf{ioco}\ \Xi(s_2) \Rightarrow i_1 \,\|\, i_2\ \mathbf{ioco}\ \Xi(s_1) \,\|\, \Xi(s_2)$$

Test restriction. A disadvantage of demonic completion is that it destroys information about underspecified behavior. On the basis of the underspecified LTS, one can conclude that traces including an unspecified input need not be tested because every implementation will always pass; after completion, however, this is no longer visible, and so automatic test generation will yield many spurious tests.

In order to avoid this, we characterize $\mathbf{ioco} \circ \Xi$ directly over LTS's. In other words, we extend the relation from $\mathcal{IOTS} \times \mathcal{IOTS}$ to $\mathcal{IOTS} \times \mathcal{LTS}$, in such a way as to obtain the same testing power but to avoid these spurious tests. For this purpose, we restrict the number of traces after which we test.

Definition 7. *Let* $s \in \mathcal{LTS}(I, U)$.

$$Utraces(s) =_{\mathrm{def}} \{\sigma \in L_\delta^* \mid s \stackrel{\sigma}{\Longrightarrow} \wedge (\nexists q', \sigma_1 \cdot a \cdot \sigma_2 = \sigma : a \in I \wedge s \stackrel{\sigma_1}{\Longrightarrow} q' \wedge q' \stackrel{a}{\not\Longrightarrow})\}$$

Intuitively, the $Utraces$ are the $Straces$ without the underspecified traces. A trace σ is underspecified if there exists a prefix $\sigma_1 \cdot a$ of σ, with $a \in I$, for which $s \overset{\sigma_1}{\Longrightarrow} q'$ and $q' \overset{a}{\not\Longrightarrow}$. We use $\mathbf{ioco}_U$ as a shorthand for $\mathbf{ioco}_{Utraces}$. In the following proposition we state that $\mathbf{ioco}_U$ is equivalent to $\mathbf{ioco} \circ \Xi$. This equivalence is quite intuitive. $\mathbf{ioco} \circ \Xi$ uses extra states to handle underspecified behavior, which are constructed so as to display chaotic behavior. If $\Xi(s)$ reaches such a state, then all behavior is considered correct. $\mathbf{ioco}_U$, on the other hand, circumvents underspecified behavior, because it uses $Utraces$.

Theorem 4. $\mathbf{ioco}_U = \mathbf{ioco} \circ \Xi$

6 Conclusions

The results of this paper imply that $\mathbf{ioco}$ can be used for compositional testing if the specifications are modeled as IOTS's; see theorems 1 and 2.

We proposed the function Ξ to complete an LTS specification; i.e., transform an LTS to an IOTS in a way that captures our notion of underspecification. This means that the above results become applicable and the $\mathbf{ioco}$ theory with completed specifications can be used for compositional testing. The resulting relation is slightly weaker than the original $\mathbf{ioco}$ relation; previously conformant implementations are still conformant, but it might be that previously non-conformant implementations are allowed under the modified notion of conformance.

Testing after completion is in principle (much) more expensive since, due to the nature of IOTS's, even the completion of a finite specification already displays infinite testable behavior. As a final result of this paper, we have presented the implementation relation $\mathbf{ioco}_U$. This relation enables us to use the original component specifications, *before* completion, for compositional testing (see theorem 4).

The insights gained from these results can be recast in terms of *underspecification*. $\mathbf{ioco}$ recognizes two kinds of underspecification: omitting input actions from a state (which implies a *don't care* if an input does occur) and including multiple output actions from a state (which allows the implementation to choose between them). It turns out that the first of these two is not compatible with parallel composition and hiding.

Testing in context. We have discussed the pre-congruence properties mainly in the context of compositional testing, but the results can easily be transposed to testing in context. Suppose an implementation under test i is tested via a context c. The tester interacts with c, and c interacts with i; the tester cannot directly interact with i. Then we have $I_i \subseteq U_c$ and $U_i \subseteq I_c$, and L_i is not observable for the tester, i.e., hidden. The tester observes the system as an implementation in a context in the following way: $\mathcal{C}[i] = \mathbf{hide}\ (I_i \cap U_c) \cup (I_c \cap U_i)\ \mathbf{in}\ c \,\|\, i$. Now theorem 1 and 2 directly lead to the following corollary for testing in context.

Corollary 2. *Let* $s, i \in \mathcal{IOTS}$ *occur in test context* $\mathcal{C}[\_]$. $\mathcal{C}[i]\ \ \mathbf{io}\not\mathbf{co}\ \mathcal{C}[s] \Rightarrow i\ \ \mathbf{io}\not\mathbf{co}\ s$

Hence, an error detected while testing the implementation in its context is a real error of the implementation, but not the other way around: an error in the implementation may not be detectable when tested in a context. This holds of course under the assumption that the test context is error free.

Relevance. We have shown a way to handle underspecification of input actions when testing communicating components with the **ioco** theory. This idea is new for LTS testing. It is inspired by [10] and work done on partial specifications in FSM testing [11].

Furthermore we have established a pre-congruence result for **ioco** for parallel composition and hiding. This is important because it shows that **ioco** is usable for compositional testing and testing in context. It establishes a formal relation between the components and the integrated system. As far as we know this result is new for both LTS testing and FSM testing. In FSM testing there are so called Communicating FSM's to model the integration of components. However we have not found any relevant research on the relation between conformance with respect to the CFSM and conformance with respect to its component FSM's.

Traditionally conformance testing is seen as the activity of checking the conformance of a single black box implementation against its specification. The testing of communicating components is often considered to be outside the scope of conformance testing. The pre-congruence result shows that the **ioco** theory can handle both problems in the same way.

Future work. The current state of affairs is not yet completely satisfactory, because the notion of composition that we require is not defined on general labeled transition systems but just on IOTS's. Testing against IOTS's is inferior, in that these models do not allow the "input underspecification" discussed above: for that reason, testing against an IOTS cannot take advantage of information about "don't care" inputs (essentially, *no* testing is required after a "don't care" input, since by definition every behavior is allowed). We intend to solve this issue by extending IOTS's with a predicate that identifies our added chaotic states. Testing can stop when the specification has reached a chaotic state.

Acknowledgments. We want to thank D. Lee and A. Petrenko for sharing their knowledge of FSM testing and for their insightful discussions.

References

1. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software—Concepts and Tools **17** (1996) 103–120
2. Tretmans, J.: Testing concurrent systems: A formal approach. In Baeten, J., Mauw, S., eds.: CONCUR'99. Volume 1664., LNCS, Springer-Verlag (1999) 46 – 65
3. Fernandez, J.C., Jard, C., Jéron, T., Viho, C.: An experiment in automatic generation of test suites for protocols with verification technology. Science of Computer Programming – Special Issue on COST247, Verification and Validation Methods for Formal Descriptions **29** (1997) 123–146
4. He, J., Turner, K.: Protocol-Inspired Hardware Testing. In Csopaki, G., Dibuz, S., Tarnay, K., eds.: Int. Workshop on Testing of Communicating Systems 12, Kluwer Academic Publishers (1999) 131–147
5. Belinfante, A., Feenstra, J., Vries, R.d., Tretmans, J., Goga, N., Feijs, L., Mauw, S., Heerink, L.: Formal test automation: A simple experiment. In Csopaki, G., Dibuz, S., Tarnay, K., eds.: 12^{th} Int. Workshop on Testing of Communicating Systems, Kluwer Academic Publishers (1999) 179 – 196

6. Petrenko, A., Yevtushenko, N.: Fault detection in embedded components. In Kim, M., Kang, S., Hong, K., eds.: Tenth Int. Workshop on Testing of Communicating Systems, Chapman & Hall (1997) 272–287
7. ISO/IEC JTC1/SC21 WG7, ITU-T SG 10/Q.8: Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, Geneve (1996)
8. Jard, C., Jéron, T., Tanguy, L., Viho, C.: Remote testing can be as powerful as local testing. In: Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE XI /PSTV XVIII '99, Kluwer Academic Publishers (1999)
9. Petrenko, A., Yevtushenko, N., Von Bochman, G.: Fault models for testing in context. In Gotzhein, R., Bredereke, J., eds.: FORTE. Volume 69 of IFIP Conference Proceedings., Kluwer (1996) 163 – 178
10. De Nicola, R., Segala, R.: A process algebraic view of Input/Output Automata. Theoretical Computer Science **138** (1995) 391–423
11. Petrenko, A., Bochmann, G.v., Dssouli, R.: Conformance relations and test derivation. In Rafiq, O., ed.: Sixth Int. Workshop on Protocol Test Systems. Number C-19 in IFIP Transactions, North-Holland (1994) 157–178
12. van der Bijl, M., Rensink, A., Tretmans, J.: Component based testing with ioco. Technical report, University of Twente (2003) URL: http://wwwhome.cs.utwente.nl/˜vdbijl/papers/CBT.pdf.
13. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. Journal of the Association for Computing Machinery **31** (1984) 560–599