Compositional Specifications for **ioco** Testing

Przemysław Daca
and Thomas A. Henzinger
IST Austria
Klosterneuburg, Austria
{przemek, tah}@ist.ac.at

Willibald Krenn
and Dejan Ničković
AIT Austrian Institute of Technology GmbH.
Vienna, Austria
{willibald.krenn, dejan.nickovic}@ait.ac.at

***Abstract*—Model-based testing is a promising technology for black-box software and hardware testing, in which test cases are generated automatically from high-level specifications. Nowadays, systems typically consist of multiple interacting components and, due to their complexity, testing presents a considerable portion of the effort and cost in the design process. Exploiting the compositional structure of system specifications can considerably reduce the effort in model-based testing. Moreover, inferring properties about the system from testing its individual components allows the designer to reduce the amount of integration testing.**

In this paper, we study compositional properties of the ioco-testing theory. We propose a new approach to composition and hiding operations, inspired by contract-based design and interface theories. These operations preserve behaviors that are compatible under composition and hiding, and prune away incompatible ones. The resulting specification characterizes the input sequences for which the unit testing of components is sufficient to infer the correctness of component integration without the need for further tests. We provide a methodology that uses these results to minimize integration testing effort, but also to detect potential weaknesses in specifications. While we focus on asynchronous models and the ioco conformance relation, the resulting methodology can be applied to a broader class of systems.

Keywords*—*compositional testing, model-based testing

I. Introduction

Modern software and hardware system design usually involves the integration of interacting components that work together in order to realize some requested behavior. Fig. 1 illustrates two components I_1 and I_2 that are composed together to form the system I. The complexity of the individual components together with their elaborate cooperation protocols often results in behavioral faults. Therefore, verification and validation methods are applied to ensure that the embedded system satisfies its specification. This is in particular true for safety-critical designs, for which correctness evidence is imposed by the regulation bodies (see for example the automotive standard ISO 26262 [1]).

Up to date, design simulation combined with testing remains the preferred technique in industry to demonstrate the correctness of software and hardware systems. Typically, verification engineers need to first test individual system components (*unit testing* of I_1 and I_2), and then test the complete system (*integration testing* of I). This process relies on verification engineers manually generating test vectors from specifications given as informal (natural language) requirements. This process is inherently time consuming, ad-hoc and prone to human errors. As a result, testing represents the main bottleneck in the design of complex systems today.

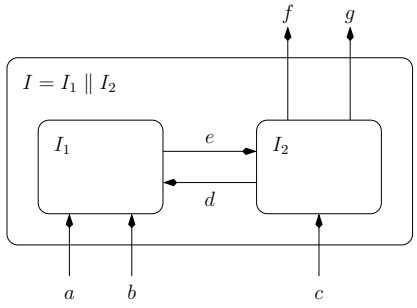


Fig. 1. A system consisting of two interacting components.

Model-based testing is a technology that provides formalization and automation to the test case generation and execution process, thus reducing time and cost of systems design. In model-based testing a *system-under-test* (SUT), denoted by I, is tested for conformance to its formal specification model S, derived from informal requirements. While S is a formal object, I is a "black-box", a physical implementation with unknown internal structure. The SUT I can be accessed by the tester only through its external interface. In order to reason about the compliance of I to S, one needs to use the *testing assumption* (see [2]), stating that I can be modeled in the same formalism as S and that I is *receptive* (*input-enabled*), i.e. it accepts all inputs at any point in time. In contrast to I, S does not need to be receptive. Lack of input (*under-specification*) in a given state of the specification models the assumption that the external environment for I does not provide that input. If the environment nevertheless emits this input action, the specification allows I to freely choose its response.

In this paper, we focus on the **ioco**-testing theory [2], a model-based testing framework for input/output labeled transition systems (IOLTS). The **ioco**-testing theory is centered around the *input/output conformance relation* **ioco**. Informally, we say that an implementation I **ioco**-conforms to its specification S if any experiment derived from S and executed on I leads to an output in I that is foreseen by S. In the **ioco**-testing theory, the lack of outputs or internal transitions is observable via a *quiescence* action.

In its original form, **ioco**-testing does not take the compositional aspects of systems into account. For example, in Fig. 1, I is the result of composing the components I_1 and I_2. Typically, the actions over which the two components synchronize (e and d in the example) are *hidden* and become unobservable to the tester after integration. In order to cope with costly testing

of large systems, results of unit testing individual components need to be used to infer properties about the composed system, to avoid or at least minimize expensive integration tests. The compositional **ioco**-testing problem can be formulated as follows: if I_1 **ioco**-conforms to its specification model S_1 and I_2 **ioco**-conforms to its specification model S_2, can we infer that I also **ioco**-conforms to S, where I and S denote the composed implementation and specification after hiding synchronization actions?

This question was first addressed in [3]. The authors show that in the general case neither parallel composition of specifications nor hiding of actions are compositional in the **ioco**-testing theory. This result is not surprising. Parallel composition is an operation tailored to receptive models. We argue that it is not an appropriate operator for composing under-specified models where one component can generate an output which is not expected as an input by the other. In addition, the hiding operation introduces partial observation over the actual state of the SUT and can result in confusion regarding the under-specified parts of the system. The authors of [3] propose two alternative restrictions to the specification models in order to preserve compositionality of **ioco**. The first option is to disallow under-specification of inputs. This is a very strong requirement in practice, since components are usually designed to operate in constrained environments. The second option allows starting with under-specification, but requires *demonic completion* of specification models — an operation which makes the assumptions about the component's environment explicit and thus makes the specification model input-enabled. We claim that demonic completion can hide important information from the tester about the poor quality of a specification and thus obscure its original intent.

We propose a different approach to compositional **ioco**-testing which, in contrast to [3], does not restrict the specification models. We define two operations — *friendly composition* and *hiding* — that are tailored to the integration of non-receptive specifications. They are based on a game-theoretic *optimistic* approach, inspired by interface theories [4], [5]. The result of the friendly composition is the overall specification that integrates the component specifications while pruning away any inputs that lead to incompatible interactions between the components. The friendly hiding operation prunes away inputs that lead to states which are ambiguous with respect to under-specification after hiding. After composing the component specifications followed by hiding of synchronizing actions, the resulting specification defines all input sequences for which no integration testing is needed — the correct integration follows from the conformance of the individual components to their specification. In addition to these technical results, this paper provides guidelines to identify specifications that are poorly modeled for compositional testing. We argue that pruned input sequences often indicate weaknesses in the specification and can be addressed by: (1) strengthening the specifications; (2) making more outputs observable; and (3) integration testing. Indeed, the proper formalization of requirements resulting in high-quality component specifications is crucial for exploiting the compositional nature of systems in testing. Investing efforts in improving models can considerably minimize expensive integration tests. We discuss methodological aspects of using our technical results to improve component models and to tailor them to compositional testing.

In Section II, we further motivate the problem of compositional testing with **ioco** and provide an informal overview of our approach, illustrating it with a vending machine example. We also identify modeling issues and discuss problems related to compositional **ioco**-testing and sketch possible solutions. Section III recalls the basics of the **ioco**-testing theory including the known results about compositional **ioco**-testing. We provide the formal presentation of our approach in Section IV and evaluate it in Section V. We present related work in more detail in Section VI, and finally conclude the paper in Section VII, giving future perspectives for our work. The proofs are presented in a separate report [6].

II. Overview and Motivating Example

In this section, we develop the example of a *drink vending machine*, which we use to (1) further motivate the problem; (2) highlight the difficulties of compositional testing within the **ioco**-theory; and (3) provide an informal overview of our proposed approach to tackle the problem.

The drink vending machine consists of the *user interface* and the *drink maker* components. The user interface specification S_1 is shown in Fig. 2 (a). S_1 requires that the user first inserts a coin (*coin?*)[1], and then selects either a tee (*utee?*) or a coffee (*ucoffee?*). After choosing the coffee, the user can also request milk (*umilk?*) for the coffee. The drink request (*mtee!*, *mcoffee!* or *mcoffeemilk!*) is forwarded to the drink maker, and the user interface waits for an acknowledgment (*done?*) that the drink was delivered to the user. When the drink is ready, the user interface emits a message (*msg!*) to the user and returns to its initial state.

The drink maker specification S_2, depicted in Fig. 2 (b), waits for a drink request (*mcoffee?* or *mcoffeemilk?*) from the user interface. Upon receiving the drink request, S_2 signals the delivery of the drink to the user (actions *coffee!* and *coffeemilk!*) and finally an acknowledgment (*done!*) is sent to the user interface. Note that S_2 is under-specified in its initial state A — it omits the action *mtee?* and thus makes an assumption that the user interface never requests a tee.

We note that in states 1 and 2 of S_1, and state A of S_2 only inputs are allowed. In the **ioco**-testing theory, such states are called *quiescent*, where the quiescence denotes the absence of observable outputs and internal actions. The absence of outputs is considered to be observable, and is marked as a special δ action (green self-loops in Fig. 2). Since quiescence is usually not explicitly modeled by the designer, we omit marking quiescent transitions in the rest of the section.

Specifications S_1 and S_2 give a certain freedom in implementing user interface and drink machine components. In particular, implementations can choose how to treat under-specified (unexpected) inputs. For instance, S_1 assumes that the user inserts a coin before choosing the drink. If the user swaps the order of actions, and first orders a drink, S_1 allows the implementation to react to this input in an arbitrary way. Fig. 3 depicts possible implementations I_1 and I_2 of their respective specifications S_1 and S_2. The user interface implementation I_1 closely follows its specification

[1]We consistently use the symbol *?* to denote an input, and the symbol *!* to denote an output action.

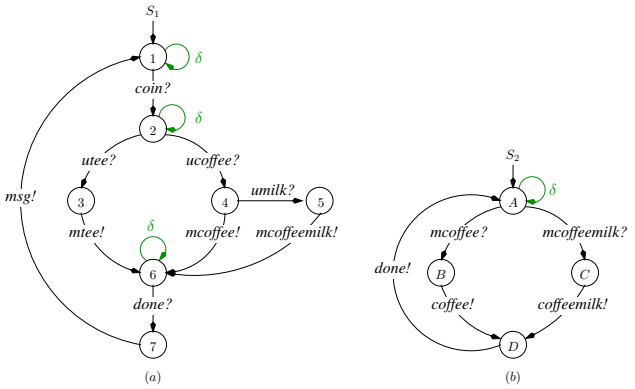


Fig. 2. Vending machine — specification of components: (a) user interface S_1; and (b) drink maker S_2.

and silently ignores all unexpected inputs (marked in blue). For example, if the user requests a coffee before inserting a coin (in state 1), this request is silently consumed by I_1. This implementation is **ioco**-conformant to its specification because it never generates an output which is not foreseen by S_1. Similarly to I_1, the drink maker implementation I_2 also silently consumes unexpected inputs in all states, except in the state A. In the initial state, I_2 reacts to a tee request (*mtee?*) by moving to the state B, from which a coffee (*coffee!*) is delivered to the user. Although preparing a coffee upon a tee request may not be a logical behavior, it is **ioco**-conformant to S_2 — the specification does not impose any particular reaction to the tee request in its initial state, giving to correct implementations complete freedom of handling such input.

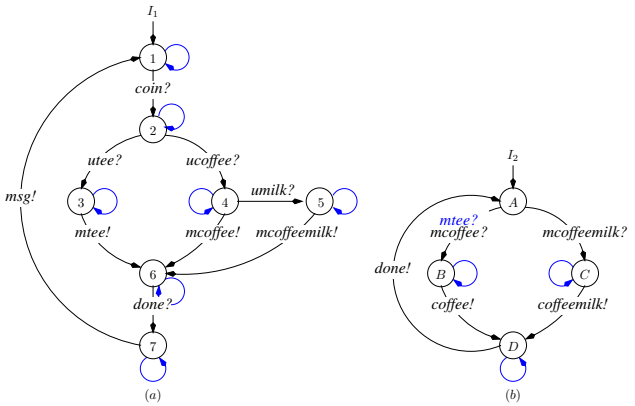


Fig. 3. Vending machine — implementation of components: (a) user interface S_1; and (b) drink maker S_2. To improve readability of the figure, we omit the missing input labels on blue self-loops. For instance, state 1 of S_1 is labeled with input actions *utee?*, *ucoffee?*, *umilk?* and *done?*, while states B, C and D of S_2 are labeled with *mtee?*, *mcoffee?* and *mcoffeemilk?*.

In the classic **ioco**-theory [3], specifications are combined with the *parallel composition* operation for input/output transitions systems [7]. Informally, parallel composition of two specifications is their Cartesian product, where each specification is allowed to take local actions independently, but the two specifications must synchronize on shared actions. Fig. 4 (a) depicts the parallel composition of S_1 and S_2, where *mtee*, *mcoffee*, *mcoffeemilk* and *done* are the shared actions. In addition to parallel composition, we may wish to *hide* shared actions, which are often only used to synchronize components, but are not observable by the external user. In the vending machine example, the user can observe inputs to the vending machine (inserting a coin and choosing the drink) and the outputs from the machine (the actual drink and the acknowledgment message). The actions *mtee*, *mcoffee*, *mcoffeemilk* and *done* are used for proper synchronization between the user interface and the drink maker components and are not visible to the user. Fig. 4 (b) depicts the parallel composition of specifications S_1 and S_2 in which the shared actions are hidden (denoted by the special action τ). Fig. 5 (a) and (b) show the parallel composition of the component implementations without and with hiding of the synchronization actions, respectively.

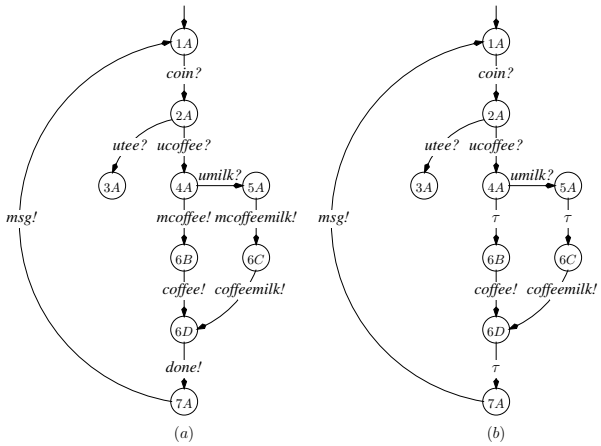


Fig. 4. Vending machine — specification: (a) parallel composition of S_1 and S_2; (b) with shared actions *mtee*, *mcoffee*, *mcoffeemilk* and *done* hidden.

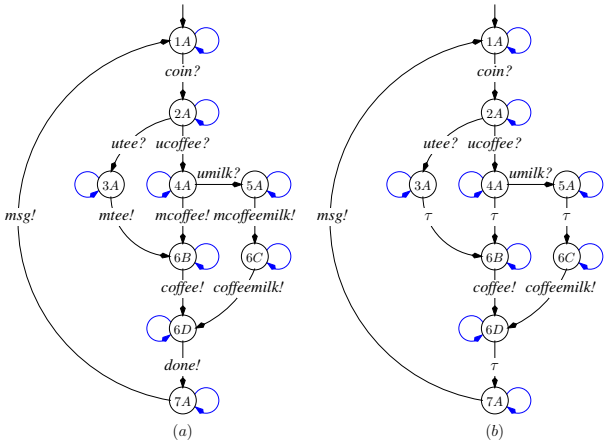


Fig. 5. Vending machine — implementation: (a) parallel composition of I_1 and I_2; (b) with shared actions *mtee*, *mcoffee*, *mcoffeemilk* and *done* hidden.

While I_1 is **ioco**-conformant to S_1 and I_2 is **ioco**-conformant to S_2, the composition of I_1 and I_2 is not **ioco**-conformant to the composition of S_1 and S_2 (with or without hiding of shared actions), as shown in [3]. We now explain the reasons for non-compositionality of **ioco** in the vending machine example.

Assuming that shared actions are not hidden, consider a test case starting with the input sequence $coin? \cdot utee?$. According to the specification (Fig. 4 (a)), the only allowed observation after reading this sequence is quiescence, since state $3A$ does not have any outgoing transitions labeled by an output action.

However, the composed implementation (Fig. 5 (a)) emits *mtee!*, an output action not allowed by the specification after reading the same sequence. It follows that **ioco**-conformance is not preserved by parallel composition.

Parallel composition is an operation tailored to combining receptive components — whenever a component outputs a shared action the other component is by definition able to consume it from any of its local states. This is not the case for non-receptive models. Indeed, S_1 emits the shared action *mtee!* in its local state 3, while the drink maker specification S_2 is not ready to consume it in its local state A, i.e. the assumption of S_2 is not fulfilled by S_1. This results in a "deadlock" state $3A$ in the parallel composition of S_1 and S_2. However, the intended meaning of under-specifying the action *mtee?* in the state A of S_2 is that S_2 is free to choose any reaction to this unexpected action. This is in contrast to what happens in state $3A$ of the composed specification.

We now consider the case when the shared actions are hidden. After reading the input sequence $coin? \cdot ucoffee?$, the state of the composed system after hiding is not uniquely defined — it can be either $4A$ or $6B$ (Fig. 5 (b)) since the user cannot observe whether the hidden action *mcoffee!* has taken place. Note that the specification (Fig. 4 (b)) leaves the input *umilk?* unspecified in $6B$, but not in $4A$, thus resulting in an ambiguity on what an external observer can expect as the reaction to this input. In fact, according to the **ioco**-theory, the only allowed observable output after executing the sequence $coin? \cdot ucoffee? \cdot umilk?$ is *coffeemilk!*, while the composed implementation can output both *coffeemilk!* (if in state $6C$) or *coffee!* (if in state $6B$).

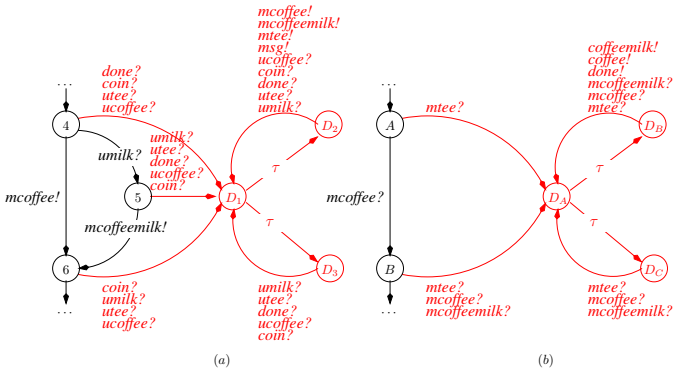


Fig. 6. Parts of the demonically completed specifications: (a) S_1; and (b) S_2. States and transitions resulting from the completion are marked in red.

In [3], the authors propose two solutions to the above anomalies. Both solutions guarantee that **ioco**-conformance is preserved by parallel composition and hiding. The first solution requires specification models to be receptive. We claim that this restriction is too strong — under-specification of inputs is one of the most powerful modeling tools for specifying open systems. A component is almost always expected to work correctly only in constrained contexts, and under-specification of inputs allows to exactly define a valid operating environment. The second solution allows non-receptive models, but requires their *demonic completion* — an operation that makes the model effectively input-enabled. Demonic completion results in adding transitions labeled with the under-specified inputs from every state to a newly inserted "sink" portion of the graph. The sink portion essentially self-loops with all inputs and outputs. Demonic completion makes the intended meaning of input under-specification explicit: when a system receives an unexpected input, it has the full freedom to choose its reaction to this input. Fig. 6 depicts parts of the demonically completed specifications S_1 and S_2.

While demonic completion preserves the intended meaning of specifications, we argue that it does not provide a fully satisfactory solution to compositional **ioco**-testing. First, the resulting specification after demonic completion increases in size. Although linear, this increase is still important for extensively under-specified models. For instance, S_1 has 7 states and 9 transitions, while its size increases to 10 states and 55 transitions after completion. Second, demonic completion obfuscates the distinction between foreseen and unspecified interactions between components. In Fig. 6 (a) the information that the input action *ucoffee?* is not expected in state 6 is lost.

This lack of distinction between foreseen and unexpected interactions between components masks the fact that we often deal with component specifications of poor quality. This results in a composition which does not faithfully represent the intended behavior of the overall system. In particular, composition with hiding of demonically completed specifications may result in many vacuous behaviors. We illustrate this problem with the example from Fig. 6. After composing demonically completed variants of S_1 and S_2, and hiding the synchronization action *mcoffee*, the external observer cannot distinguish between states $4A$ and $6B$, which in contrast to the original composition (see Fig. 4 (b)), now both admit the input action *umilk?*. However, *umilk?* triggers a transition from $6B$ to the state D_1B, from which the demonically completed specification S_1 allows all possible behaviors, including the one in which the acknowledgment message is sent back to the user without any drink being served.

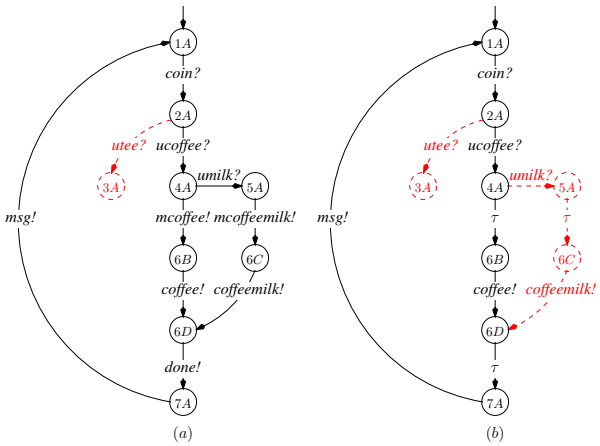


Fig. 7. Vending machine — overall specification: (a) friendly composition of S_1 and S_2; (b) with friendly hiding of synchronization actions *mtee*, *mcoffee*, *mcoffeemilk* and *done*.

We propose an alternative approach to composition and hiding that provides additional information to the designer. It is inspired by interface theories [4], [5]. We provide a new composition operation, called the *friendly composition*, which takes an *optimistic* approach to combining two specifications. Following the optimistic approach, two specifications are compatible for composition, if there exists *some* context in which

they can interact while both satisfying their guarantees. We have seen in Fig. 4 (a) that the interaction of two specifications (user interface and drink maker) results in states in which one component is allowed to emit an output which is not expected by the other component (action *mtee*! in the state $3A$). We declare such states as *ambiguous states*, and compute the *maximal* compound environment which avoids such states. The algorithm that computes the friendly composition of two specifications prunes all states from which the environment cannot prevent the composed system from reaching an ambiguous state. The resulting composite specification combines the compatible interactions of two component specifications. Fig. 7 (a) depicts the friendly composition of S_1 and S_2, where red dashed transitions and states are the ones pruned away from their parallel composition.

Similarly to friendly composition, we define the *friendly hiding* operation, which prunes away from the specification with hidden actions all the states that can become ambiguous when interacting with an external environment. The resulting specification is depicted in Fig. 7 (b), where red dashed transition and states are pruned away from the composite specification of S_1 and S_2 after hiding.

The main technical contribution of this paper is the definition of the friendly composition and friendly hiding operations, for which we show that they preserve **ioco**-conformance. In contrast to combining demonic completion with parallel composition and hiding, our approach results in composite specifications of smaller size. The resulting composite specification defines behaviors for which no integration testing is needed. Apart from the technical contribution, we argue that friendly composition and hiding expose weaknesses of component specifications to the designer. The pruned behaviors after applying friendly composition and hiding indicate that assumptions made by individual component specifications may be too weak and deserve more careful analysis. We claim that the time spent on improving the quality of the component specifications so that they allow compositional testing is rewarded with the avoidance of integration tests. We distinguish between the following scenarios in using our approach to derive better compositional specifications.

Scenario A: the designer can guarantee that the composed system will be used in the context defined by the assumptions of the friendly-composed specification and that the ambiguous interactions will never take place. In this case, no additional integration testing is needed. In the vending machine example, this would mean that the designer has the possibility of disabling the tee and milk request buttons.

Scenario B: by hiding shared actions information about the internal state of the SUT may be lost, resulting in ambiguous states. Better observability can be achieved by keeping some shared actions visible to the external environment. From the technical point of view, keeping the *mcoffee!* action visible in the vending machine specification allows for compositional testing.

Scenario C: the designer cannot guarantee that the composed system will be used in the context defined by the friendly-composed specification assumptions. In this case, the specification is too weak and needs a revision. In our example, the race between the input action *umilk?* and the output action *mcoffee!* in S_1 indicates a poor specification. We propose a different specification, which requires an additional action and in which the user is expected to make all requests before the machine is able to process them.

III. Preliminaries

In this section, we define labeled transition systems, parallel composition and hiding operations, input/output conformance relation (**ioco**), and recall the previous results on compositional properties of **ioco**.

A. Labeled Transition Systems

An *input/output labeled transition system* (IOLTS) is a formal model for specifying reactive systems. An IOLTS A is a tuple $(Q, L^I, L^O, T, \hat{q})$, where Q is a countable set of *states*, L^I and L^O are disjoint countable sets of *input* and *output labels*, $\hat{q} \in Q$ is the *initial state* and T is the *transition relation*. We denote by $L = L^I \cup L^O$ the set of all labels of an IOLTS. To avoid ambiguity we may use subscripts, like Q_A, to indicate that an element belongs to an IOLTS A. We consider IOLTS with possibly *silent* transitions, denoted by τ, hence the transition relation is defined as $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$. An IOLTS is said to be *receptive*, denoted by R-IOLTS, if for all $q \in Q$ and for all $a \in L^I$, there exists an outgoing transition from q labeled by a. For instance, specifications S_1 and S_2 in Fig. 2 are IOLTS, while implementations I_1 and I_2 in Fig. 3 are R-IOLTS. *Strongly-convergent* IOLTS are transition systems that do not have loops consisting of only silent transitions. We use the standard abbreviated notation, where $\mu \in L \cup \{\tau\}$ and $a \in L$

$$
\begin{array}{lcl}
q \xrightarrow{\mu} q' & \equiv & (q, \mu, q') \in T \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' & \equiv & \exists q_0, \ldots, q_n \text{ st. } q = q_0 \xrightarrow{\mu_1} q_1 \\
 & & \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} q_n = q' \\
q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & \equiv & \exists q' \text{ st. } q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \not\xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & \equiv & \neg \exists q' \text{ st. } q \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} q' \\
q \stackrel{\epsilon}{\Rightarrow} q' & \equiv & q = q' \text{ or } q \xrightarrow{\tau \cdot \ldots \cdot \tau} q' \\
q \stackrel{a}{\Rightarrow} q' & \equiv & \exists q_1, q_2 \text{ st. } q \stackrel{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \stackrel{\epsilon}{\Rightarrow} q' \\
q \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q' & \equiv & \exists q_0, \ldots, q_n \text{ st. } q = q_0 \stackrel{a_1}{\Rightarrow} q_1 \\
 & & \stackrel{a_2}{\Rightarrow} \ldots \stackrel{a_n}{\Rightarrow} q_n = q' \\
q \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} & \equiv & \exists q' \text{ st. } q \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q' \\
q \not\stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} & \equiv & \neg \exists q' \text{ st. } q \stackrel{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q'
\end{array}
$$

A sequence $\sigma \in (L \cup \{\tau\})^+$ is an *execution* of an IOLTS A if $\hat{q}_A \xrightarrow{\sigma}$. A sequence $\sigma \in L^*$ is a trace of A if $\hat{q}_A \stackrel{\sigma}{\Rightarrow}$. We denote by $\mathbf{Traces}(A)$ the set of all traces of A. The sequence *coin* $\cdot$ *ucoffee* $\cdot$ τ $\cdot$ *coffee* is an execution of the specification shown in Fig. 4 (b), while *coin* $\cdot$ *ucoffee* $\cdot$ *coffee* is its trace. Given a subset of labels $L' \subseteq L$ and σ a sequence over L, we denote by $\sigma_{\downarrow L'}$ the projection σ to the set of labels L'.

We say that a state $q \in Q$ of A is *quiescent*, denoted by $\delta(q)$, if it has no outgoing output or internal actions. Quiescent states emit a special *quiescence* action δ, which indicates that A cannot proceed without input from the environment. The *suspension automata* are IOLTS, where quiescent actions are made explicit. Formally, given an IOLTS $A = (Q, L^I, L^O, T, \hat{q})$, its suspension automaton A_δ is the IOLTS

$A_\delta = (Q, L^I, L^O \cup \{\delta\}, T \cup T_\delta, \hat{q})$, where $T_\delta = \{q \xrightarrow{\delta} q \mid \delta(q)\}$. We denote by **Straces**(A) the set $\{\sigma \in (L \cup \{\delta\})^* \mid \hat{q} \stackrel{\sigma}{\Rightarrow}_{A_\delta}\}$ of traces of A_δ, also called *suspension traces*. Specifications S_1 and S_2 shown in Fig. 2 show explicitly quiescence actions, where for example $\delta \cdot \mathit{coin} \cdot \delta$ is in **Straces**(S_1).

B. Parallel Composition

Two components can be integrated if their input/output actions do not conflict. In particular, we require that the intersection of their input (output) label sets is empty. Formally, we say that two IOLTS A_1 and A_2 are *composable* if $L_1^I \cap L_2^I = L_1^O \cap L_2^O = \emptyset$. When two composable IOLTS are composed, they synchronize on shared actions and move independently on other actions. Formally, *parallel composition* is defined as follows.

Definition 1 (Parallel composition): Let $A_1 = (Q_1, L_1^I, L_1^O, T_1, \hat{q}_1)$ and $A_2 = (Q_2, L_2^I, L_2^O, T_2, \hat{q}_2)$ be two composable IOLTS. Their *parallel composition*, denoted by $A_1 \parallel A_2$, is the IOLTS $(Q_{1\|2}, L_{1\|2}^I, L_{1\|2}^O, T_{1\|2}, \hat{q}_{1\|2})$, where $Q_{1\|2} = Q_1 \times Q_2$, $L_{1\|2}^I = (L_1^I \backslash L_2^O) \cup (L_2^I \backslash L_1^O)$, $L_{1\|2}^O = L_1^O \cup L_2^O$, $\hat{q}_{1\|2} = (\hat{q}_1, \hat{q}_2)$, and $T_{1\|2}$ is defined by the rules:

$$\frac{q_1 \xrightarrow{\mu}_{A_1} q_1' \quad \mu \in (L_1 \cup \{\tau\}) \backslash L_2}{(q_1, q_2) \xrightarrow{\mu}_{A_{1\|2}} (q_1', q_2)}$$

$$\frac{q_2 \xrightarrow{\mu}_{A_2} q_2' \quad \mu \in (L_2 \cup \{\tau\}) \backslash L_1}{(q_1, q_2) \xrightarrow{\mu}_{A_{1\|2}} (q_1, q_2')}$$

$$\frac{q_1 \xrightarrow{\mu}_{A_1} q_1' \quad q_2 \xrightarrow{\mu}_{A_2} q_2' \quad \mu \in L_1 \cap L_2}{(q_1, q_2) \xrightarrow{\mu}_{A_{1\|2}} (q_1', q_2')}$$

The specification $S_1 \parallel S_2$ shown in Fig. 4 (a) represents the parallel composition of specifications S_1 and S_2.

C. Hiding

The parallel composition of two components is often followed by *hiding* some of the actions on which they synchronize. We follow the process algebraic approach in which parallel composition and hiding operations are two separate operations, and formally define hiding as follows.

Definition 2 (Hiding): Let $A = (Q, L^I, L^O, T, \hat{q})$ be an IOLTS and $\Sigma \subseteq L^O$ be the subset of output actions. The *hiding* of Σ in A, denoted by $h_\Sigma(A)$, is the tuple $(Q, L^I, L^O \setminus \Sigma, h_\Sigma(T), \hat{q})$, where $h_\Sigma(T)$ is obtained from T by replacing every transition $(q, a, q') \in T$ labeled by an output action $a \in \Sigma$ by the transition (q, τ, q').

The specification $h_\Sigma(S_1 \parallel S_2)$ shown in Fig. 4 (b) represents the hiding of Σ in $S_1 \parallel S_2$, where $\Sigma = \{\mathit{mtee}, \mathit{mcoffee}, \mathit{mcoffeemilk}, \mathit{done}\}$.

D. Input/Output Conformance Relation

Given an IOLTS A, the set **out**$(q) \equiv \{a \in L^O \mid q \xrightarrow{a}\ \} \cup \{\delta \mid \delta(q)\}$ is the set of all outputs (including δ if q is quiescent) that are defined when the system is in state q. The set q **after** $\sigma \equiv \{q' \mid q \stackrel{\sigma}{\Rightarrow}_{A_\delta} q'\}$ denotes the set of states that can be reached in A from q after reading σ in its suspension automaton A_δ. We now present the formal definition of the **ioco** relation.

Definition 3: Given a R-IOLTS I and an IOLTS S, we say that I **ioco** S iff

$$\forall \sigma \in \textbf{Straces}(S),\ \textbf{out}(\hat{q}_I \textbf{ after } \sigma) \subseteq \textbf{out}(\hat{q}_S \textbf{ after } \sigma).$$

In the vending machine example, both I_1 **ioco** S_1 and I_2 **ioco** S_2, where S_1 and S_2 are depicted in Fig. 2 (a) and (b), and I_1 and I_2 are depicted in Fig. 3 (a) and (b), respectively. We now recall the results from [3] which state that **ioco** is not preserved in general under parallel composition and hiding, but is preserved if all the specifications are receptive. Receptiveness of specifications can be achieved by demonic completion (see [3] for its formal definition).

Theorem 1 ([3]): Given two composable R-IOLTS I_1 and I_2, two composable IOLTS S_1 and S_2 and two composable R-IOLTS S_1^* and S_2^*, we have

$$\begin{array}{lcl} I_1 \textbf{ ioco } S_1 \wedge I_2 \textbf{ ioco } S_2 & \not\rightarrow & (I_1 \parallel I_2) \textbf{ ioco } (S_1 \parallel S_2) \\ I_1 \textbf{ ioco } S_1^* \wedge I_2 \textbf{ ioco } S_2^* & \rightarrow & (I_1 \parallel I_2) \textbf{ ioco } (S_1^* \parallel S_2^*). \end{array}$$

Theorem 2 ([3]): Given a R-IOLTS I, an IOLTS S and a R-IOLTS S^*, defined over the alphabet L, and a subset $\Sigma \subseteq L^O$, we have

$$\begin{array}{lcl} I \textbf{ ioco } S & \not\rightarrow & h_\Sigma(I) \textbf{ ioco } h_\Sigma(S) \\ I \textbf{ ioco } S^* & \rightarrow & h_\Sigma(I) \textbf{ ioco } h_\Sigma(S^*). \end{array}$$

Consider the vending machine example, in which I_1 **ioco** S_1 and I_2 **ioco** S_2, the sequence $\sigma = \mathit{coin} \cdot \mathit{utee}$ and the compositions of specifications and implementations before and after hiding (see Fig. 4 and 5). The available output after executing σ on $I_1 \parallel I_2$ is *mcoffee*, while the composed specification $S_1 \parallel S_2$ allows only δ after executing the same sequence, hence $I_1 \parallel I_2$ does not **ioco**-conform to $S_1 \parallel S_2$. After hiding the actions $\Sigma = \{\mathit{mtee}, \mathit{mcoffee}, \mathit{mcoffeemilk}, \mathit{done}\}$ in the composition, we obtain additional traces which are not **ioco**-conformant. For instance, after executing the sequence $\mathit{coin} \cdot \mathit{ucoffee} \cdot \mathit{umilk}$ in $h_\Sigma(I_1 \parallel I_2)$, the possible outputs are *coffee* and *coffeemilk*, while the specification $h_\Sigma(S_1 \parallel S_2)$ allows only the action *coffeemilk* after executing the same sequence.

We note that demonic completion introduces new "chaotic" states to the original specification, from which all behaviors are allowed. Exploring chaotic states in test case generation is not useful. In order to avoid their exploration [3] defines the set of **Utraces**. Intuitively, **Utraces** restricts **Straces** by eliminating underspecified traces. We obtain the **uioco** conformance relation by replacing **Straces** with **Utraces** in the definition of **ioco**. However, it turns out that **uioco** does not preserve compositional properties (see [6] for details).

IV. Optimistic Approach to Composition and Hiding

In this section, we formalize the *friendly* composition and hiding operations, presented informally in Section II.

A. Friendly Environments

We have seen in Section III that parallel composition and hiding can introduce *ambiguous* states, and that **ioco** is not preserved under these two operations. An *ambiguous* state results from the parallel composition of two IOLTS in which one emits an action that the other one is not ready to accept.

Definition 4 (Ambiguous state): Given two composable IOLTS A and B, a pair $(q_A, q_B) \in Q_A \times Q_B$ is an *ambiguous state* if there exists a shared action $\alpha \in L_A \cap L_B$ such that either: (1) $\alpha \in L_A^O$, $q_A \xrightarrow{\alpha}$ and $q_B \not\xrightarrow{\alpha}$; or (2) $\alpha \in L_B^O$, $q_B \xrightarrow{\alpha}$ and $q_A \not\xrightarrow{\alpha}$.

In the parallel composition $S_1 \parallel S_2$ of the vending machine example, depicted in Fig. 4 (a), the state $3A$ is ambiguous because S_1 emits the output action *mtee!*, while S_2 does not accept it.

Inspired by contract-based design and interface theories, we propose an optimistic approach to composition and hiding. In this optimistic setting, we look for a *friendly environment* which steers the specification away from ambiguous states. A friendly environment is helpful towards the systems, by always accepting the system's outputs and never providing actions that the system cannot accept as inputs.

Definition 5 (Friendly environment): Given an IOLTS $A = (Q, L^I, L^O, T, \hat{q})$, a *friendly environment* for A is a strongly-convergent IOLTS $E = (Q_E, L^O, L^I, T_E, \hat{q}_E)$ such that $A \parallel E$ does not have ambiguous states.

A *composition-friendly* environment does not allow a composed system to reach an ambiguous state in the composition.

Definition 6 (Composition-friendly environment): Given a composed IOLTS $A \parallel B$, a *composition-friendly environment* for $A \parallel B$ is its friendly environment such that for all ambiguous states $(q_A, q_B) \in Q_{A\parallel B}$, for all $q_E \in E$, $((q_A, q_B), q_E)$ is not reachable in $(A \parallel B) \parallel E$.

A friendly environment is *maximal* if it admits more behaviors than any other friendly environment. The maximal friendly environment is used to compute the largest portion of the specification that is guaranteed to preserve conformance under composition and hiding. The resulting specification characterizes all sequences for which integration testing is not necessary.

Definition 7 (Maximal friendly environment): A friendly environment E for an IOLTS A is said to be *maximal* if for all friendly environments E' for A it holds $\mathbf{Traces}(E') \subseteq \mathbf{Traces}(E)$.

The IOLTS fragment that interacts correctly with its friendly environment E is called its E-reachable fragment. It is obtained by composing the IOLTS with E, while keeping the original meaning of inputs and outputs.

Definition 8 (Environment reachable fragment): Let A be an IOLTS and E be its friendly environment. The *E-reachable fragment of A* is an IOLTS $(Q, L^I, L^O, T, \hat{q})$, where $Q = Q_{A\parallel E}$, $\hat{q} = \hat{q}_{A\parallel E}$, $L^I = L_A^I$, $L^O = L_A^O$, and $T = T_{A\parallel E}$.

B. Friendly Composition, Friendly Hiding and *ioco*

We are now ready to formally define *friendly composition* and *hiding*, and show that **ioco** is a pre-congruence for these two operations. We use the maximal friendly environment to restrict the classical parallel composition and hiding operations to a fragment that guarantees avoiding ambiguous states.

Definition 9 (Friendly composition): Given two composable IOLTS A and B, we say that they are *compatible* if there exists a composition-friendly environment E for $A \parallel B$. Given two compatible IOLTS A and B, their *friendly composition*, denoted by $A \otimes B$, is an E-reachable fragment of $A \parallel B$, where E is the maximal composition-friendly environment for $A \parallel B$.

Lemma 1: For any two compatible IOLTS A and B, there exists a maximal composition-friendly environment for $A \parallel B$.

Definition 10 (Friendly hiding): Given an IOLTS A and $\Sigma \subseteq L^O$, the *friendly Σ-hiding* of A, denoted by $\hat{h}_\Sigma(A)$, is an E-reachable fragment of $h_\Sigma(A)$, where E is the maximal friendly environment for $h_\Sigma(A)$.

The specifications $S_1 \otimes S_2$ and $\hat{h}_\Sigma(S_1 \otimes S_2)$ depicted in Fig. 7 (a) and (b) represent friendly composition of S_1 and S_2, followed by the friendly hiding of the synchronization actions $\Sigma = \{mtee, mcoffee, mcoffeemilk, done\}$.

We note that for receptive models, the friendly composition and friendly hiding coincide with the parallel composition and hiding. We state the main technical contributions of the paper — that **ioco** relation is preserved under friendly composition and friendly hiding.

Theorem 3: Given two compatible R-IOLTS I_1 and I_2 and two compatible IOLTS S_1 and S_2, we have

$$I_1 \textbf{ ioco } S_1 \wedge I_2 \textbf{ ioco } S_2 \rightarrow (I_1 \otimes I_2) \textbf{ ioco } (S_1 \otimes S_2).$$

Theorem 4: Given a R-IOLTS I and an IOLTS S defined over the set L^O of output labels and $\Sigma \subseteq L^O$, we have

$$I \textbf{ ioco } S \rightarrow \hat{h}_\Sigma(I) \textbf{ ioco } \hat{h}_\Sigma(S).$$

Corollary 1: Given two compatible R-IOLTS I_1 and I_2, two compatible IOLTS S_1 and S_2, and $\Sigma \subseteq L^O_{1\parallel 2}$, we have

$$I_1 \textbf{ ioco } S_1 \wedge I_2 \textbf{ ioco } S_2 \rightarrow \hat{h}_\Sigma(I_1 \otimes I_2) \textbf{ ioco } \hat{h}_\Sigma(S_1 \otimes S_2).$$

C. Computing Friendly Composition and Hiding

In this section, we present the algorithms for effectively computing the friendly composition and hiding. We first create the *deterministic* maximal friendly environment E for an IOLTS A. It is constructed by swapping input and output labels of A and applying a variant of the subset construction that determinizes the IOLTS afterwards.

Our variant of the subset construction works as follows. Consider an IOLTS A, the standard subset construction C of A, and a state S of C. When C is in S, the maximal environment must accept all output transitions from S. In contrast, the environment provides an input to C only if all states in S can accept it. As we can see, E quantifies existentially over outputs in A and universally over inputs in A.

Definition 11 (Maximal deterministic friendly environment): The *maximal deterministic friendly environment* of an IOLTS $A = (Q, L^I, L^O, T, \hat{q})$ is an IOLTS $E_{\max}(A) = (2^Q, L^O, L^I, T_E, \hat{Q})$, where $\hat{Q} = \hat{q}$ **after** ϵ,

and $(S, \alpha, S') \in T_E$ if: 1) $S' = S$ **after** α; 2) $\alpha \in L^I$ implies that $q \xrightarrow{\alpha}$ for all $q \in S$; and 3) $\alpha \in L^O$ implies that there exists $q \in S$ such that $q \xrightarrow{\alpha}$.

Algorithm 1 Friendly composition

Input: composable IOLTS A_1 and A_2
Output: $A_1 \otimes A_2$ or not compatible
  $E \leftarrow E_{\max}(A_1 \parallel A_2)$
  Amb $\leftarrow$ states in E that contain an ambiguous s. of $A_1 \parallel A_2$
  Prune $\leftarrow \emptyset$
  for all $S \in$ *Amb* **do**
    Prune $\leftarrow$ *Prune* $\cup$ $\{$state S' in $E \mid \exists \sigma \in {L^O_{A_1 \parallel A_2}}^* : S' \stackrel{\sigma}{\Rightarrow}_E S\}$
  remove states in *Prune* from E
  if E has no initial state **then return** not compatible
  else return E-reachable fragment of $(A_1 \parallel A_2)$

Algorithm 1 constructs the friendly composition for IOLTS A_1 and A_2 or returns the information that they are not compatible. First, it constructs the maximal deterministic friendly environment $E_{\max}(A_1 \parallel A_2)$. The algorithm then computes the set *Amb* of states in $E_{\max}(A_1 \parallel A_2)$ that contain an ambiguous state from $A_1 \parallel A_2$ and prunes away all states in $E_{\max}(A_1 \parallel A_2)$ that reach *Amb* by a trace of output labels in $A_1 \parallel A_2$. If the initial state is removed in the process, then no friendly environment for $A_1 \parallel A_2$ exists. Otherwise, E is the resulting maximal composition-friendly environment for $A_1 \parallel A_2$ and the E-reachable fragment of $A_1 \parallel A_2$ is their friendly composition. The friendly composition $A_1 \otimes A_2$ constructed by the algorithm is of size at most $|A_1| \cdot |A_2| \cdot 2^{|A_1| \cdot |A_2|}$ and if A_1 and A_2 are both deterministic, then $A_1 \otimes A_2$ is of size $|A_1| \cdot |A_2|$.

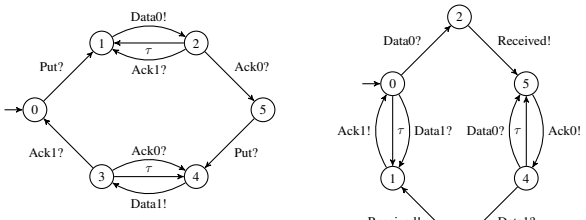
Algorithm 2 computes the friendly Σ-hiding of an IOLTS A, where $\Sigma \subseteq L^O_A$. To obtain the maximal deterministic friendly environment, we simply hide actions in A, determinize it and then construct its maximal friendly environment E. The friendly hiding $\hat{h}_\Sigma(A)$ computed by the algorithm is of size $2^{|A|}$ and if A is deterministic, then $\hat{h}_\Sigma(A)$ is of size $|A|$. We note that there always exists a friendly environment E for arbitrary A. It suffices that E accepts all outputs from A and does not provide any inputs.

Algorithm 2 Friendly hiding

Input: IOLTS A, $\Sigma \subseteq L^O_A$,
Output: $\hat{h}_\Sigma(A)$
  $E \leftarrow E_{\max}(h_\Sigma(A))$
  return E-reachable fragment of $h_\Sigma(A)$

V. Evaluation

To evaluate our approach to compositional testing in the **ioco**-theory, we implemented a proof-of-concept tool for computing friendly composition and hiding. We applied the tool to the *alternating bit protocol* taken from the CADP examples repository [8]. We were interested in the quality of the component specifications with respect to compositional testing. We also compared the size of the friendly composition of the component specifications to the size of the specification obtained by the demonic completion approach.



(a) Sender A (b) Receiver B

Fig. 8. CADP specifications of the sender and receiver component for a variant of the Alternating Bit-Protocol. The τ transitions model time-outs.

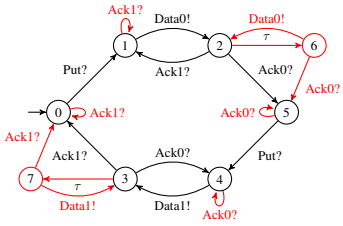
The alternating bit protocol is a data transfer protocol supporting re-transmission of lost or corrupted messages. For the sake of simplicity, we assume a perfect link (no corruption of messages). The protocol consists of a sender A and a receiver B. Initially, the sender A waits for data to be transmitted (*Put?*). Upon receiving the data, it sends it to B together with a sequence bit (*Data0!*) and waits for an acknowledgment (*Ack0?*). If the acknowledgment does not arrive before a time-out, A re-transmits the data. This behavior is modeled by a τ-transition, which abstracts away the actual time-out value. Once A receives the acknowledgment, it flips the sequence bit and repeats the procedure (*Data1!* and *Ack1?*).

Receiver B behaves in a similar way. It first waits for data marked with the sequence bit (*Data0?*) and possibly takes a time-out transition. Upon receiving the data, it sends it to the external environment (*Put!*) and, in a next step, sends an acknowledgment marked with the same sequence bit to the sender (*Data0!*). The receiver then repeats the procedure with the flipped sequence bit.

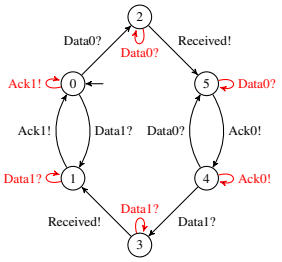
Sender A and receiver B (cf. Fig. 8) are not compatible, i.e. no friendly environment guarantees the correctness of the protocol. There is a number of composition-ambiguous states in the parallel composition of A and B, mainly due to the time-out (τ) transitions. For instance, A can be either in state 1 or 2 after reading the trace *Put*$\cdot$*Data0*. In state 2, A expects the input *Ack0?* which is not the case in state 1, where A is ready to re-transmit *Data0!* and is brought to after a time-out. Similar problems occur with the receiver B due to its own time-out transitions.

It follows that the specifications for A and B are too weak for compositional testing. In order to strengthen the specification of A, we need to improve the handling of the race between re-transmitting data to B and receiving the acknowledgment from B. We tackle the problem by making the assumptions about the handling of an acknowledgment more explicit and introduce additional states in the specification. The similar time-out problem of receiver B is handled in a slightly different way: time-out is no longer modeled as a τ-transition but as a self-loop that allows B continuous re-transmission of acknowledgments to A, while waiting for new data. The strengthened specifications A' and B' of the sender and receiver are depicted in Fig. 9 (a) and (b). Their friendly composition (Fig. 9 (c)) contains no composition-ambiguous state.

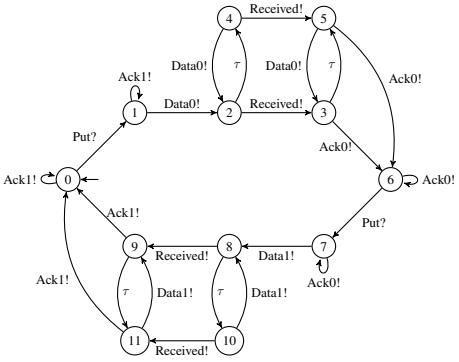
Hiding all synchronization actions (*Data* and *Ack*) in



(a) Sender A'

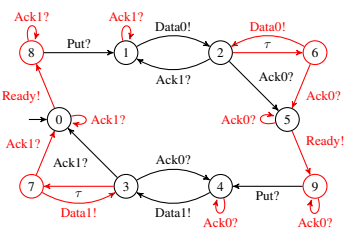


(b) Receiver B'

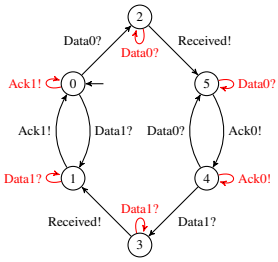


(c) $A' \otimes B'$

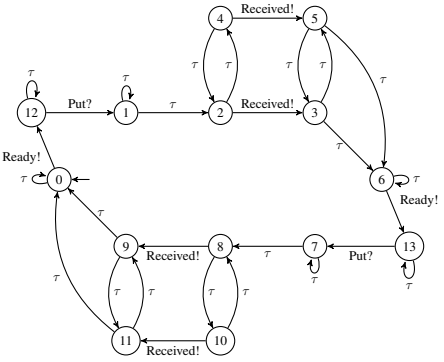
Fig. 9. Strengthened specifications (a) A'; (b) B'; and (c) their friendly composition.



(a) Sender A''



(b) Receiver B'



(c) $\hat{h}_\Sigma(A'' \otimes B')$

Fig. 10. Strengthened specifications (a) A''; (b) B'; and (c) their friendly composition followed by friendly hiding of Σ.

Model	# tran	# states
A	10	6
B	10	6
$d(A)$	31	9
$d(B)$	28	9
A''	22	10
B'	14	6
$h_\Sigma(d(A) \parallel d(B))$	76	34
$\hat{h}_\Sigma(A'' \otimes B')$	24	12

TABLE I. SIZES OF SPECIFICATION MODELS AND THEIR COMPOSITIONS.

$A' \otimes B'$ introduces new ambiguous states. A friendly environment cannot observe the internal state of $A' \otimes B'$ and decide when the protocol is ready to receive new data items (*Put?* action). The easy way to overcome this problem is to add constraints to the original specification of the sender that say how to handle the input *Put?* when the sender is in a non-observable state. The solution we use in this example, however, is to strengthen the sender specification by adding an output action *Ready!* which tells the external environment that it is ready to accept new data. The resulting specification A'' is depicted in Fig. 10 (a). The new specification requires a hand-shake between the protocol and the environment and results in the friendly composition $\hat{h}_\Sigma(A'' \otimes B')$ followed by the friendly hiding of $\Sigma = \{Ack0, Ack1, Data0, Data1\}$. The composite specification $\hat{h}_\Sigma(A'' \otimes B')$ does not encounter any ambiguous states. It follows that any implementation of sender A'' and receiver B' can be tested individually and that their composition is correct-by-construction, without need for additional integration tests.

We finally compare the size of $\hat{h}_\Sigma(A'' \otimes B')$ to the one of $h_\Sigma(d(A) \parallel d(B))$, where $d(A)$ and $d(B)$ denote the demonically completed variants of A and B. The results are shown in Table I. We first observe that by applying our approach we obtain specifications of smaller size than by demonically completing the component specifications and then applying parallel composition and hiding. This is in particular visible when comparing the size of $h_\Sigma(d(A) \parallel d(B))$ (76 transitions and 34 states), to the one of $\hat{h}_\Sigma(A'' \otimes B')$ (24 transitions and 12 states). We note that in our approach, only foreseen interactions between components are taken into account, which is not the case with the demonic completion approach. While our framework may require manual improvement of the component specifications, we argue that this is the right procedure to arrive at specifications of good quality for compositional testing. Although more automated, the demonic completion approach to compositional testing admits many useless implementations.

To summarize, the case study shows that the alternating bit protocol specification was not modeled with compositional testing in mind. Parts of the sender and receiver specifications are not sufficiently specified for cooperative interactions and do not admit compositional testing with **ioco**. We improved the specifications by strengthening the assumptions where needed. We note that despite the strengthening, the improved specifications are not input-enabled.

We finally remark that although the individual component specifications are strongly convergent, their composition with hiding is not. This may be a problem for testing with quiescence in general (see [2]) but does not affect our work as presented here: we only require friendly environments to be strongly-convergent.

VI. RELATED WORK

This paper is inspired by [3] and extends it by defining new composition and hiding operations adapted for under-

specified models and preserving compositional properties of **ioco**. Compositional properties of the real-time conformance relation **tioco** were studied in [9]. In order to preserve compositional testing with **tioco**, specifications are required to be receptive. Compositional properties of the **cspio** conformance relation for model-based testing with CSP specifications were studied in [10]. CSP operations are shown to be monotonic with respect to **cspio** when the specifications are input-enabled, or the implementations are not receptive, and after each trace, the input actions accepted by the implementation are a subset of those offered by the specification. The compositional testing problem for systems modeled as networks of abstract components, based on coalgebraic definitions, was considered in [11]. Once again, specifications must be receptive to preserve compositional properties in testing. Assume-guarantee reasoning is combined with **ioco** in [12] in order to allow compositional testing. This work is complementary to ours, as it starts from a global specification of the complete system, and uses assumptions about components to divide and conquer the testing process. A methodology to reduce the efforts of integration testing is presented in [13]. It combines model-based integration with model-based testing, but does not provide formal arguments that support the proposed approach.

This paper is also inspired by the interface theories [4], [5]. In contrast to interface automata, used in the context of contract-based design, this work focuses on compositional properties in testing. Instead of iterative design through stepwise refinement, the **ioco**-theory assumes the existence of an implementation. We consider **ioco** with its explicit treatment of quiescence as the refinement relation, rather than alternation simulation used in interface automata. The integration of specifications in the **ioco**-theory separates parallel composition from hiding, thus allowing for multicast and broadcast communication. In interface automata, parallel composition and hiding are combined into a single operation, thus allowing point-to-point communication only. We also mention similar frameworks in contract-based design: synchronous interfaces with and without shared variables [14], synchronous relational interfaces [15], and real-time interfaces [16], [17].

VII. Conclusion and Future Perspectives

We proposed a novel approach to compositional testing for **ioco** based on friendly composition and hiding. Our framework characterizes foreseen interactions between components and minimizes the effort needed for integration testing. In addition to the technical results, this paper gives new insights to compositional testing in general and the associated difficulties. In particular, we use our approach to provide guidelines for identifying weaknesses in component specifications and improving them with compositional testing in mind. Since high-level specifications are typically much smaller than the actual implementations, we argue that this additional effort in model analysis is rewarded with a reduction of effort in expensive integration testing.

In our framework, we assume that the composition of component specifications is the specification of the integrated system. In the future, we will study how to exploit our results when the overall system is specified with a separate model. In particular, we will investigate whether our results can be combined with [12]. In addition, we will study whether we can weaken the notion of the ambiguous states, while preserving compositional properties of **ioco**. Although we considered asynchronous models and **ioco**-theory, we are confident that our results can be adapted to different modeling frameworks and conformance relations. In fact, many issues related to compositional testing come from the power of the IOLTS model, where components compete in executing the actions without much restrictions. We will adapt our work to the synchronous data-flow systems, which we believe have more robust properties with respect to composition and hiding.

Acknowledgments

This research was funded in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM), by the ARTEMIS JU under grant agreement numbers 269335 (MBAT) and 295373 (nSafeCer), and by the Austrian Science Fund (FWF) project S11402-N23 (RiSE).

References

[1] "ISO/DIS 26262-1 - Road vehicles Functional safety Part 1 Glossary," Geneva, Switzerland, Tech. Rep., Jul. 2009.

[2] J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[3] M. van der Bijl, A. Rensink, and J. Tretmans, "Compositional testing with ioco," in *FATES*, ser. LNCS, vol. 3395. Springer, 2003, pp. 86–100.

[4] L. de Alfaro and T. A. Henzinger, "Interface automata," in *ESEC / SIGSOFT FSE*, 2001, pp. 109–120.

[5] ——, "Interface theories for component-based design," in *EMSOFT*, 2001, pp. 148–165.

[6] P. Daca, T. A. Henzinger, W. Krenn, and D. Ničković, "Compositional specifications for ioco testing: Technical report," IST Austria, Technical Report, 2014, http://repository.ist.ac.at/152/.

[7] N. A. Lynch and M. R. Tuttle, "Hierarchical correctness proofs for distributed algorithms," in *PODC*, 1987, pp. 137–151.

[8] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: a toolbox for the construction and analysis of distributed processes," *STTT*, vol. 15, no. 2, pp. 89–107, 2013.

[9] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Formal Methods in System Design*, vol. 34, no. 3, pp. 238–304, 2009.

[10] A. Sampaio, S. Nogueira, and A. Mota, "Compositional verification of input-output conformance via csp refinement checking," in *ICFEM*, ser. LNCS, vol. 5885. Springer, 2009, pp. 20–48.

[11] M. Aiguier, F. Boulanger, and B. Kanso, "A formal abstract framework for modelling and testing complex software systems," *Theor. Comput. Sci.*, vol. 455, pp. 66–97, 2012.

[12] L. B. Briones, "Assume-guarantee reasoning with ioco testing relation," in *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers*, 2010, pp. 103–107.

[13] N. C. W. M. Braspenning, J. M. van de Mortel-Fronczak, and J. E. Rooda, "A model-based integration and testing method to reduce system development effort," *Electr. Notes Theor. Comput. Sci.*, vol. 164, no. 4, pp. 13–28, 2006.

[14] A. Chakrabarti, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang, "Synchronous and bidirectional component interfaces," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 414–427.

[15] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee, "A theory of synchronous relational interfaces," *ACM Trans. Program. Lang. Syst.*, vol. 33, no. 4, p. 14, 2011.

[16] A. David, K. G. Larsen, A. Legay, U. Nyman, and A. Wasowski, "Timed i/o automata: a complete specification theory for real-time systems," in *HSCC*. ACM, 2010, pp. 91–100.

[17] L. de Alfaro, T. A. Henzinger, and M. Stoelinga, "Timed interfaces," in *EMSOFT*, ser. LNCS, vol. 2491. Springer, 2002, pp. 108–122.