Complete Composition Operators for ioco-Testing Theory

Nikola Beneš
Faculty of Informatics
Masaryk University
Brno, Czech Republic
xbenes3@fi.muni.cz

Przemysław Daca
IST Austria
Klosterneuburg, Austria
przemek@ist.ac.at

Thomas A. Henzinger
IST Austria
Klosterneuburg, Austria
tah@ist.ac.at

Jan Křetínský
IST Austria
Klosterneuburg, Austria
jan.kretinsky@ist.ac.at

Dejan Ničković
AIT Austrian Institute of
Technology GmbH
Vienna, Austria
dejan.nickovic@ait.ac.at

ABSTRACT

We extend the theory of input-output conformance with operators for merge and quotient. The former is useful when testing against multiple requirements or views. The latter can be used to generate tests for patches of an already tested system. Both operators can combine systems with different action alphabets, which is usually the case when constructing complex systems and specifications from parts, for instance different views as well as newly defined functionality of a previous version of the system.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods

Keywords

ioco; Model-based testing; Decomposition; Specification merging

1. INTRODUCTION

Development and verification of modern hardware and software systems faces numerous challenges due to their complexity. For safety-critical applications, regulation bodies impose rigorous standards that require convincingly showing the absence of behavioral faults. There is a wide selection of verification and validation (V&V) techniques to support demonstration of the designed system's correctness. These techniques range from model checking to manual testing, but many of them suffer from a number of drawbacks. For instance, despite the tremendous progress in model checking research and practice over the past decades, scalability remains an issue for modern systems. In addition, model checking is typically used to check the correctness of an abstract model of the design, but usually does not address its actual implementation. At the other extreme, manual testing remains the preferred V&V method in industry. While manual testing provides a practical way to check the correctness of an implementation, it remains an informal, tedious, ad-hoc and error-prone activity with uncertain coverage. It is widely recognized that despite much progress, V&V remains the main bottleneck in the design of complex systems, and hence the need for new rigorous and systematic, but also pragmatic and scalable verification and testing techniques.

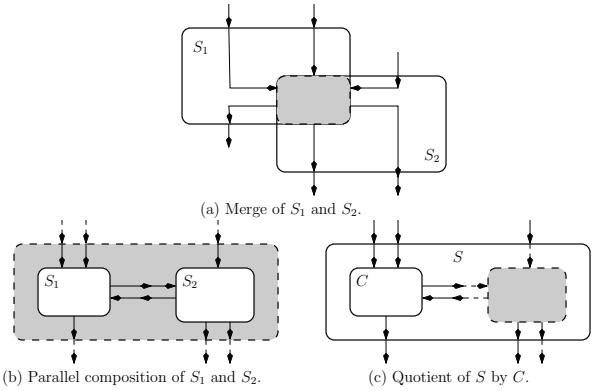
Model-based testing (MBT), also known as "black-box" testing, provides a promising compromise between formal verification and manual testing. Similarly to model checking, MBT uses an abstract model of the system-under-test (SUT). This model faithfully represents the core behavior of the system, while hiding some less significant implementation details. This abstract model is then used to automate the generation of test cases in a systematic way. In particular, the generated test suite is typically guaranteed to meet some coverage criteria, thus providing additional assurance about the quality of tests. The generated test cases are then executed on the actual SUT, checking its compliance to the abstract model. MBT techniques require a conformance relation between the abstract model and the SUT. For instance, the **ioco** conformance relation is at the core of the de-facto standard MBT theory for input/output labeled transition systems. Informally, we say that a physical implementation i **ioco**-conforms to its abstract model s if any test generated from s and executed on i leads to a response by i that is foreseen by s.

The specification and design of complex systems can benefit from a compositional flow that makes the testing activity more effective. Such a flow needs to be supported by operations that provide structure both to the specification and the system. The *merge* operation, illustrated in Figure 1 (a), enables natural structuring of specifications into conjunctions of requirements, where each requirement addresses a specific aspect that the system needs to satisfy. The *parallel composition*, illustrated in Figure 1 (b), allows one to structure a system as a network of interacting sub-systems. Finally, the *quotient*, depicted in Figure 1 (c), is an operation dual to the parallel composition, which formalizes the notion of design "patching" and enables the synthesis of an unknown sub-system specification from the specification of the overall system and of the other components. Given a specification s of the overall system and c of a subset of its components and their interactions, the quotient s/c defines the part of the

CBSE'15, May 4–8, 2015, Montréal, QC, Canada.

http://dx.doi.org/10.1145/2737166.2737175.

system-wide specification that needs to be satisfied by the remaining part of the implementation.



(a) Merge of S_1 and S_2.

(b) Parallel composition of S_1 and S_2.

(c) Quotient of S by C.

Figure 1: Illustration of merge, parallel composition and quotient operations. The results of the operations are depicted in gray.

These composition operations have been studied mostly as a theoretical foundation of specification theories. In this paper, we propose to apply them in the setting of the **ioco**-testing theory. Since they enable exploiting the structural properties of both the system and its specification, the testing effort could be reduced in practice.

The operation of merge acts as a logical conjunction. Firstly, given a specification s, we can check whether requirements $s_1, \ldots, s_n$ (obtained for instance manually) are actually a decomposition of s by constructing their merge and comparing it to s. Then we can generate tests more efficiently with respect to s_i's instead of achieving coverage of the entire specification s. Secondly, given partial specifications (views) $s_1, \ldots, s_n$, we can check for their consistency. We construct their merge and show it is empty if and only if the partial specifications are contradictory.

The parallel composition, denoted by $\|$, has already been studied for **ioco**. It has been shown that given subsystems i_1, i_2 **ioco**-conforming to their respective specifications s_1, s_2, the overall system $i_1 \parallel i_2$ **ioco**-conforms to $s_1 \parallel s_2$. Thus the properties of the parallel composition enable inferring properties of the overall system from sub-system (unit) testing and thus minimize costly integration testing. In our paper, we use parallel composition in order to define the quotient.

Given the specification s of the overall system and the specification of a sub-system c, we construct their quotient. It is the most general specification that describes all subsystems that when put in parallel composition with c satisfy the specification s. As a consequence, the quotient operation allows us to improve regression testing by enabling generation of test cases that exercise only the modified behavior of an SUT after its patching.

Further, in contrast to the classical **ioco** theory, in which the specification and the SUT must be defined over the same inputs and outputs, we allow partial specifications that refer only to a subset of the SUT's alphabet. Hence, we equip the theory with another operation, called *alphabet equalization*, which completes the external interface of a partial model and makes it consistent with the SUT's interface. The proposed theory facilitates dividing the testing activity into smaller problems and minimizing the overall testing effort.

This paper focuses on merge, quotient, and alphabet equalization, and complements the results of [5, 16] on parallel composition. We summarize our main contributions:

1. A definition of and an algorithm to construct the *merge* for the **ioco** theory, allowing us to check the consistency of views as well as to construct complete models from partial models.

2. The first complete algorithm for computing the *quotient* for the **ioco** theory, allowing us to generate tests for patches of an already tested system.

3. Extending the **ioco**-theory with a notion of *alphabet equalization*, which enables true partial modeling of requirements. Consequently, we can compute merges and quotients of systems with different action alphabets, which is the case with different views as well as when defining new functionality of a previous version of the system.

1.1 Related Work

Compositional aspects of systems have received considerable attention in the past decade, especially in the context of contract-based design. Interface automata [6, 7] provide support for independent implementability and stepwise refinement via decomposition of systems into sub-systems with parallel composition. There are no known merge, quotient, and alphabet equalization operations for interface automata. The merge operation, also called shared refinement, was introduced in [8] to synchronous interfaces, and its properties were used to develop an incremental test generation procedure in [1]. Assume/Guarantee Contracts [4] provide support for parallel composition, merge, and alphabet equalization, but there is no known quotient operation. In the context of modal transition systems [10] and modal interfaces [13], parallel composition, merge, and alphabet equalization were studied in [2, 13, 14], while quotient was proposed in [12].

In the context of **ioco**-testing theory, parallel composition was first studied in [16], where the input-enabledness of specifications was required to preserve compositional properties. This work was extended in [5], where the input-enabledness requirement was dropped by introducing a composition with pruning. The quotient operation in the **ioco**-testing theory was proposed in [11]. In contrast to this paper, the proposed solution is incomplete and may not find the quotient even when it exists. In addition, the authors only consider the parallel composition operation in which synchronization actions are automatically hidden. We provide more flexibility by separating parallel composition from hiding. Finally, there is an assumption in [11] that the output alphabet of the known component specification not shared with the alphabet of the overall system specification must be used for the synchronization with the quotient. In contrast, we provide more flexibility by allowing specifying beforehand the input alphabet of the quotient, thus allowing more control over synchronization between the quotient and the known component. We are not aware of any work on merge and alphabet equalization in the context of **ioco**-testing.

2. DEFINITIONS AND PRELIMINARIES

In this section, we present the standard notions of the **ioco** theory. First, the implementation under test as well as the specification are reactive systems of the same kind. For algorithmic reasons, all sets are considered finite.

Definition 1. An *input-output labeled transition system (IOLTS)* is a tuple (Q, I, O, T, s) where Q is a set of states, I is a set of input labels, O is a set of output labels, $T \subseteq Q \times (I \cup O \cup \{\tau\}) \times Q$ is the transition relation with the silent action $\tau \notin I \cup O$, and $s \in Q$ is the initial state.

An IOLTS is an *input-output transition system (IOTS)* if each state is input enabled, i.e. $\forall q \in Q\ \forall i \in I : q \xrightarrow{i}$. (See notation in Fig. 2.)

We often abuse the notation and identify IOLTS with their initial states. While specifications can be any IOLTS, not describing behavior under certain inputs, implementations are assumed to be IOTS, being able to deal with any input. Input actions are marked in drawings with "?" and output actions with "!". The silent action τ is not visible to the observer. Unless specified otherwise, we require that IOLTS are *strongly-convergent*, i.e. they do not have infinite sequences of τ transitions.

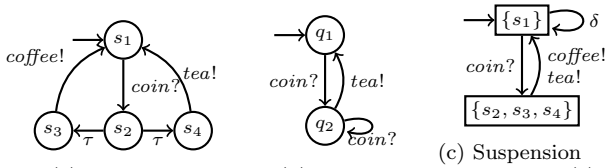
$$
\begin{array}{lcl}
q \xrightarrow{\lambda} q' & \equiv & (q, \lambda, q') \in T \\
q \xrightarrow{\lambda} & \equiv & \exists q' : q \xrightarrow{\lambda} q' \\
q \not\xrightarrow{\lambda} & \equiv & \text{not } q \xrightarrow{\lambda} \\
q \stackrel{\epsilon}{\Rightarrow} q' & \equiv & q = q' \text{ or } q \xrightarrow{\tau \cdots \tau} q' \\
q \stackrel{\ell}{\Rightarrow} q' & \equiv & \exists q_1, q_2 : q \stackrel{\epsilon}{\Rightarrow} q_1 \xrightarrow{\ell} q_2 \stackrel{\epsilon}{\Rightarrow} q' \\
q \stackrel{\delta}{\Rightarrow} q' & \equiv & q \stackrel{\epsilon}{\Rightarrow} q' \text{ such that } \forall x \in O \cup \{\tau\} : q' \not\xrightarrow{x} \\
q \xRightarrow{\alpha_1 \cdots \alpha_n} q' & \equiv & \exists q_0, \ldots, q_n : q = q_0 \xRightarrow{\alpha_1} q_1 \\
 & & \xRightarrow{\alpha_2} \ldots \xRightarrow{\alpha_n} q_n = q' \\
q \xRightarrow{\alpha_1 \cdots \alpha_n} & \equiv & \exists q' : q \xRightarrow{\alpha_1 \cdots \alpha_n} q' \\
q \not\xRightarrow{\alpha_1 \cdots \alpha_n} & \equiv & \text{not } q \xRightarrow{\alpha_1 \cdots \alpha_n}
\end{array}
$$

Figure 2: Notation for transitions in IOLTS. Here λ ranges over $I \cup O \cup \{\tau\}$, ℓ over $I \cup O$ and α's over $I \cup O \cup \{\delta\}$.

The lack of outputs and silent transitions in a state is observable as special action δ, called *quiescence*. A state q is quiescent, denoted by $\delta(q)$, if $\forall x \in O \cup \{\tau\} : q \not\xrightarrow{x}$. This gives rise to the *suspension traces* of s:

$$STraces(s) = \{\sigma \in (I \cup O \cup \{\delta\})^* \mid s \stackrel{\sigma}{\Rightarrow}\},$$

where quiescence can be observed, as opposed to usual traces $Traces(s) = STraces(s) \cap (I \cup O)^*$.



(a) IOLTS s. (b) IOTS i. (c) Suspension automaton $\Gamma(s)$.

Figure 3: Illustration of Example 1.

The input-output conformance relation **ioco** is defined between an input-enabled implementation i and a specification s over the same alphabet of inputs and outputs. Intuitively, i **ioco**-conforms to s, written $i \,\mathbf{ioco}\, s$, if after every suspension trace of s, the output actions of i are foreseen by s.

Definition 2. Given IOTS i and IOLTS s, we say that $i \,\mathbf{ioco}\, s$ if

$$\forall \sigma \in STraces(s) :\ \mathsf{out}(i \,\mathsf{after}\, \sigma) \subseteq \mathsf{out}(s \,\mathsf{after}\, \sigma),$$

where

- for $q \in Q, \sigma \in (I \cup O \cup \{\delta\})^*$: $q \,\mathsf{after}\, \sigma = \{q' \mid q \stackrel{\sigma}{\Rightarrow} q'\}$,
- for $S \subseteq Q$: $\mathsf{out}(S) = \bigcup_{q \in S} \Big(\{x \in O \mid q \xrightarrow{x}\} \cup \{\delta \mid \delta(q)\}\Big)$.

Example 1. The IOLTS in Figure 3a shows a specification s of a vending machine that accepts a coin (*coin*?) and then makes an internal choice between preparing two types of beverages (*coffee*! or *tea*!). Note that state s_1 is quiescent, so the transition $s_1 \stackrel{\delta}{\Rightarrow} s_1$ is possible. Figure 3b shows an IOTS i that is a possible implementation of s. The component i has less functionality than s, since it may only prepare tea. It is easy to check that i **ioco** s.

An IOLTS p is called *deterministic*, if for every suspension trace σ the set $p \,\mathsf{after}\, \sigma$ is a singleton. In particular, there are no silent actions in deterministic IOLTS. We often transform an IOLTS to a deterministic form where, moreover, quiescence is explicit. Consequently, the result of the transformation is *non-blocking*, i.e. for each state q we have $q \xrightarrow{x}$ for some $x \in O \cup \{\delta\}$.

Definition 3. A *suspension automaton (SA)* is a deterministic non-blocking IOLTS with output labels containing δ. Given an IOLTS (Q, I, O, T, s), the *suspension automaton of* s is the suspension automaton $(2^Q \setminus \emptyset, I, O \cup \{\delta\}, \Gamma(T), \Gamma(s))$, where $\Gamma(s) = \{s' \in Q \mid s \stackrel{\epsilon}{\Rightarrow} s'\}$ and transitions are given by

$$r \xrightarrow{a} \{q' \in Q \mid \exists q \in r : q \stackrel{a}{\Rightarrow} q'\} \text{ for } a \in I \cup O$$
$$r \xrightarrow{\delta} \{q \in r \mid \delta(q)\}$$

Figure 3c show the suspension automaton $\Gamma(s)$ of the IOLTS s of Figure 3a.

The following lemma shows that the transformation of any IOLTS to a suspension automata preserves the behavior as well as **ioco**-conformance.

LEMMA 1. *For every IOTS i and IOLTS s*

- $STraces(s) = STraces(\Gamma(s)) = Traces(\Gamma(s))$
- $i \,\mathbf{ioco}\, s \iff i \,\mathbf{ioco}\, \Gamma(s)$

2.1 Demonic completion

The merge and quotient operations described in the following sections are defined for suspension automata that are input-enabled. Every suspension automaton can be made input-enabled by *demonic completion*, i.e. adding missing input transitions to a "universal system" of two states $\{u, \bar{u}\}$ (shown in Figure 4) that accepts every legal behavior [16].

Formally, given a SA $(Q, I, O \cup \{\delta\}, T, s)$, the *demonic completion of* s is the input-enabled SA $(Q \cup \{u, \bar{u}\}, I, O \cup \{\delta\}, T', \Xi(s))$, where $T' = T \cup \{u \xrightarrow{\ell} u \mid \ell \in I \cup O\} \cup \{\bar{u} \xrightarrow{\ell} u \mid \ell \in I\} \cup \{u \xrightarrow{\delta} \bar{u}, \bar{u} \xrightarrow{\delta} \bar{u}\} \cup \{q \xrightarrow{\ell} u \mid q \in Q, \ell \in I, q \not\xrightarrow{\ell}\}$.

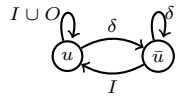


Figure 4: The universal system $\{u, \bar{u}\}$

Demonic completion preserves **ioco**-conformance, i.e. for every SA s and IOTS i, we have that

$$i \,\mathbf{ioco}\, s \iff i \,\mathbf{ioco}\, \Xi(s).$$

The universal system might also appear in the result of merge or quotient. Checking **ioco**-conformance against the

universal state is vacuous, thus we prefer to remove the universal state as a post-processing step after these operations. We use the following algorithm on an suspension automata $(Q, I, O \cup \{\delta\}, T, s)$ to remove spurious transitions to u: (1) traverse all states to check if s contains the universal system $\{u, \bar{u}\}$; (2) if $u, \bar{u}$ exist in s, then remove from T all input transitions leading to u; and (3) remove $u, \bar{u}$ if they are unreachable from s.

Given SA s, let SA q be the result of applying the algorithm above to s. Then we have for every IOTS i that

$$i \,\mathbf{ioco}\, s \iff i \,\mathbf{ioco}\, q.$$

2.2 Validity

In the following section, we provide constructions for the operations of merging and quotienting specifications. The operations will be defined on suspension automata that are "valid," meaning they represent some real IOLTS specification. However, the results of the operations are suspension automata that might not be valid. In order to fix this, we introduce a construction that transforms a possibly invalid suspension automaton to a valid one while preserving the set of its *implementations*, i.e. IOTS that **ioco**-conform to it.

As discussed in [17], several requirements must hold to ensure validity of the suspension automaton $(Q, I, O \cup \{\delta\}, T, s)$. Here we present a slightly different, but equivalent formulation of these conditions:

non-blocking $\forall q \in Q\ \exists x \in O \cup \{\delta\} : q \xrightarrow{x}$,

anomaly freedom $\forall q \xrightarrow{\delta} q'\ \forall x \in O : q' \not\xrightarrow{x}$,

quiescence observation

$$\forall q \xrightarrow{\delta} q' \xrightarrow{\delta} q'' : \mathit{Traces}(q) \supseteq \mathit{Traces}(q') = \mathit{Traces}(q'').$$

The non-blocking condition requires that every state without outputs must have a quiescent transition; however a state can have a quiescent transition even in the presence of outputs. Note that we require non-blocking for all suspension automata by definition. Anomaly freedom requires that a quiescent transition must lead to a state without any outputs. Finally, quiescence observation requires that only the first observation of quiescence may provide information about the current state, namely that there are no outputs. Further observation of quiescence do not yield any additional information (system cannot produce more outputs if it could not before and nothing was observed since then).

A suspension automaton satisfying all these conditions is called *valid*. For any IOLTS s, the suspension automaton $\Gamma(s)$ of s is valid. Further, whenever a suspension automaton s is valid, there is an (effectively constructible [17]) IOLTS $\bar{s}$ such that s and $\Gamma(\bar{s})$ have the same set of implementations, i.e. for all IOLTS i : $i \,\mathbf{ioco}\, s \iff i \,\mathbf{ioco}\, \Gamma(\bar{s})$.

3. OPERATIONS

In this section, we define merging and quotient on valid input-enabled SA and provide algorithms to compute them. In general, these operators require post-processing in order to ensure validity of the result. After merging, it is sufficient to perform *pruning*, which removes states arising from inconsistent requirements in different views. For quotient we provide an operator valid that satisfies for all (possibly invalid) suspension automata s that $\mathsf{valid}(s)$ is a valid suspension automaton with the same set of implementations as s. Formally, $\mathsf{valid}(s)$ satisfies the following axiom: for every IOTS i

$$\mathsf{valid}(s) \text{ is valid and } (i \,\mathbf{ioco}\, s \iff i \,\mathbf{ioco}\, \mathsf{valid}(s)). \quad \textbf{(V)}$$

If s does not have any implementations, then $\mathsf{valid}(s)$ is the empty suspension automaton. Therefore, for any (possibly invalid) suspension automaton that has implementations we can construct a corresponding IOLTS specification. As a result, we ensure we can generate tests from the results of the operations in the standard way [15].

The validization procedure consists of sequence of several steps that also include pruning, see Fig. 5. Interestingly, these extra steps make use of the newly defined merge operation.

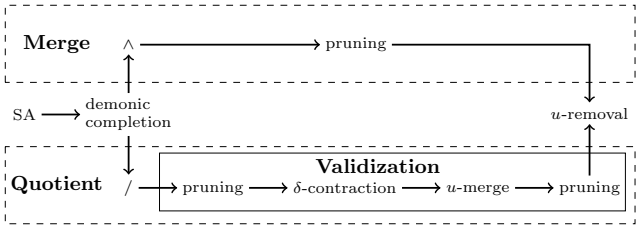


Figure 5: Flowchart for merge and quotient.

Although both merge and quotient are operations on IOLTS, they are defined in terms of operations on states of these IOLTS. Therefore, instead of writing e.g. $S_1 \wedge S_2$ for the merge of systems S_1, S_2, we write $s_1 \wedge s_2$ where s_1, s_2 are their initial states, respectively.

3.1 Merge (Conjunction)

We provide the operator of merge (conjunction) of valid suspension automata $s_1, \ldots, s_n$ producing a suspension automaton $\bigwedge_{k=1}^{n} s_k$ followed by pruning. The desired property of the merge operator is formalized as the following axiom: for every IOTS i

$$i \,\mathbf{ioco} \bigwedge_{k=1}^{n} s_k \iff \forall k \in \{1, \ldots, n\} : i \,\mathbf{ioco}\, s_k. \quad \textbf{(M)}$$

In other words, the set of implementations of the conjunction is the intersection of the respective implementation sets. The inspiration for the merge algorithm comes from the merge and common implementation algorithms for modal transition systems [3, 9]. Here we deal with systems over the same input and output labels; for different alphabets, see Section 3.3.

For suspension automata $(Q_k, I, O \cup \{\delta\}, T_k, s_k)$, $k = 1, \ldots, n$, we define the suspension automaton $(Q, I, O \cup \{\delta\}, T, \bigwedge_{k=1}^{n} s_k)$ as follows:

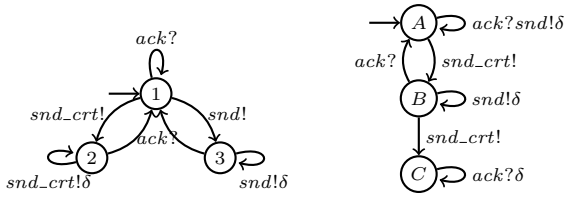
- states are given by the Cartesian product: $Q = \prod_{k=1}^{n} Q_k$,
- the initial state is $\bigwedge_{k=1}^{n} s_k = (s_1, \ldots, s_n)$,
- for each $x \in I \cup O \cup \{\delta\}$ there is a transition $(q_1, \ldots, q_n) \xrightarrow{x} (q_1', \ldots, q_n')$ whenever $q_k \xrightarrow{x} q_k'$ for all $k = 1, \ldots, n$.

This construction may give us an invalid automaton, as illustrated in the following example.

Example 2. We demonstrate the merge operator on two views of a task dispatcher module. The functional view in Figure 6a allows the dispatcher to send a task on a regular channel (*snd*!) or on a critical channel (*snd_crt*!). Once a task has been sent, the dispatcher can keep sending tasks

on the same channel until it receives an acknowledgment *ack* or output quiescence.

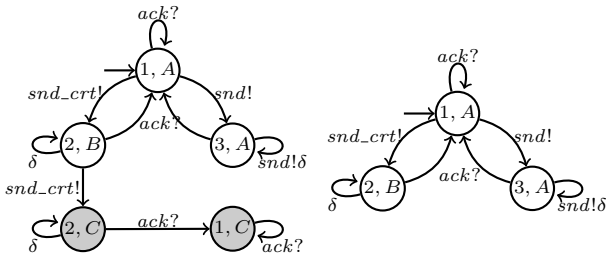
The safety view in Figure 6b requires that the system cannot dispatch any new tasks after using the critical channel twice consecutively. This view, however, does not restrict the usage of the regular channel.



(a) Functional view s_{fun}. (b) Safety view s_{safe}.

Figure 6: The task dispatcher module.

Figure 7a shows the merge of the functional and safety views before pruning. The state $(1, C)$ is blocking, because the views require contradicting behavior: the safety view does not allow the state to dispatch new tasks, while the functional view forbids it from emitting quiescence. Note that the state $(2, C)$ is also invalid, since input *ack*? leads it to a blocking state. The final merged specification after pruning is shown in Figure 7b.



(a) Before pruning. (b) After pruning.

Figure 7: Merge of the views in Figures 6a and 6b.

Pruning

To obtain a valid result, blocking states, such as $(1, C)$ of the previous example, need to be pruned. Moreover, pruning these states away may result in new blocking states, which must be pruned as well until this procedure stabilizes.

Formally, a state $q \in Q$ is invalid if

- q is blocking, i.e. there is no $x \in O \cup \{\delta\}$ with $q \xrightarrow{x}$,
- $q \xrightarrow{a?} q'$ where $a \in I$ and q' is invalid,
- for all $x \in O \cup \{\delta\}$: $q \xrightarrow{x} q'$ where q' is invalid.

The pruning thus proceeds as follows:

1. Identify all blocking states and add them to the set $\mathcal{P}$.
2. Find all states with an input transition to $\mathcal{P}$ and add them to $\mathcal{P}$.
3. Find all states such that all their output and δ transitions go to $\mathcal{P}$ and add them to $\mathcal{P}$.
4. Repeat steps 2 and 3 until no more states found.
5. Set $Q' = Q \setminus \mathcal{P}$. Leave the transitions induced by Q'.

The merge operator is given by the $\wedge$-construction followed by pruning.

THEOREM 1. *Let $(s_k)_k$ be a finite sequence of valid suspension automata. The merge operator satisfies the axiom* **(M)**. *In particular, the initial state $\bigwedge_{k=1}^{n} s_k$ is pruned if and only if there is no IOTS i with i **ioco** s_k for all s_k.*

3.2 Parallel Composition & Quotient

Parallel Composition

The compositional architecture of both implementations and specifications is achieved by the use of the parallel composition operator $\|$. We first recall this standard operator. Two IOLTS are said to be *composable* if they have no common input or output actions. The only shared actions thus belong to the set of input actions of one IOLTS and to the set of output actions of the other. Such actions are combined in the parallel composition in a synchronizing manner. It is usual to define the parallel composition without implicit hiding, i.e. the synchronization actions remain visible. The hiding of these actions can be then done explicitly, which allows to describe a variety of architectural choices. We postpone the operation of hiding to the next section.

Definition 4. Let $(Q_1, I_1, O_1, T_1, s_1)$ and $(Q_2, I_2, O_2, T_2, s_2)$ be two IOLTS such that $I_1 \cap I_2 = O_1 \cap O_2 = \emptyset$. Their parallel composition is the IOLTS $(Q, I, O, T, s_1 \parallel s_2)$ where $Q = Q_1 \times Q_2$, $O = O_1 \cup O_2$, $I = I_1 \cup I_2 \setminus O$, $s_1 \parallel s_2 = (s_1, s_2)$ and the transition relation is defined as follows:

- if $q_1 \xrightarrow{\alpha} q_1'$ for $\alpha \notin (I_2 \cup O_2)$ then $(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2)$,
- if $q_2 \xrightarrow{\alpha} q_2'$ for $\alpha \notin (I_1 \cup O_1)$ then $(q_1, q_2) \xrightarrow{\alpha} (q_1, q_2')$,
- if $q_1 \xrightarrow{\alpha} q_1'$, $q_2 \xrightarrow{\alpha} q_2'$ for $\alpha \neq \tau$ then $(q_1, q_2) \xrightarrow{\alpha} (q_1', q_2')$.

For input-enabled specifications, the parallel composition satisfies the so-called independent implementability criterion [16].[1] Let s_1, s_2 be composable input-enabled suspension automata. For every IOTS i_1, i_2

$$i_1 \textbf{ ioco } s_1,\ i_2 \textbf{ ioco } s_2 \implies i_1 \parallel i_2 \textbf{ ioco } s_1 \parallel s_2. \qquad \textbf{(PC)}$$

Observe that the other implication does not hold and the composition is thus semantically incomplete, as opposed to our merge and quotient.

Quotient

We provide the operator of *quotient*, which is dual to parallel composition. Given valid suspension automata s, c specifying the whole system and the known component, respectively, the *quotient of s by c* is the construction s/c followed by validization. Intuitively, it is the most general specification describing all components that when put in parallel with the known component c satisfy the specification s. It can thus be seen as a kind of specification decomposition. Formally, the quotient axiom is the following: for every IOTS i

$$i \textbf{ ioco } s/c \iff \forall j \textbf{ ioco } c : i \parallel j \textbf{ ioco } s. \qquad \textbf{(Q)}$$

[1]The constraint of input-enabled specifications can be dropped when a parallel composition with pruning inspired by interface automata is used [5].

Consequently, the quotient does not exist if and only if there is no i satisfying the right hand-side.

Clearly, there are more solutions to the equation depending on the input alphabet of the quotient $I_{s/c}$. In order to obtain a unique solution, our quotient operator is parameterized by $I_{s/c}$. Let $(Q_s, I_s, O_s \cup \{\delta\}, T_s, s)$ and $(Q_c, I_c, O_c \cup \{\delta\}, T_c, c)$ be suspension automata and let $I_{s/c}$ be an alphabet. We assume that the following is satisfied: $I_{s/c} \cap I_c = \emptyset$, $I_{s/c} \setminus O_c \subseteq I_s$, $O_c \subseteq O_s$, $I_s \cap O_c = \emptyset$, $I_s \subseteq I_c \cup I_{s/c}$. We first set $O_{s/c} = O_s \setminus O_c$ to be the output alphabet of the quotient. The quotient is the suspension automaton $(Q, I_{s/c}, O_{s/c} \cup \{\delta\}, T, s/c)$, defined as follows:

- The set of states is $Q = 2^{Q_s \times Q_c}$. We also define the following auxiliary sets for each $q \in Q$:
  - $next_a(q) = \{(t', d') \mid \exists (t, d) \in q : t \xrightarrow{a} t'$ and if $a \in I_c \cup O_c \cup \{\delta\}$ then $d \xrightarrow{a} d'$ else $d' = d\}$.
  - asynchronous closure $cl(q) = \{(t', d') \mid \exists (t, d) \in q : \exists \sigma \in \big((I_c \cup O_c) \setminus (I_{s/c} \cup O_{s/c})\big)^* : t \xrightarrow{\sigma} t', d \xrightarrow{\sigma} d'\}$.

  Furthermore, a state q is called *inconsistent* if there exists $a \in O_c$ and $(t, d) \in q$ such that $d \xrightarrow{a}$ and $t \not\xrightarrow{a}$.
- The initial state is $s/c = cl(\{(s, c)\})$.
- The transition relation T is defined for consistent states only:
  - for $a \in I_{s/c}$: $q \xrightarrow{a} cl(next_a(q))$,
  - for $a \in O_{s/c}$: if $\forall (t, d) \in q : t \xrightarrow{a}$ then $q \xrightarrow{a} cl(next_a(q))$,
  - for δ: if $\forall (t, d) \in q : (t \xrightarrow{\delta}$ or $d \not\xrightarrow{\delta})$ then $q \xrightarrow{\delta} cl(next_\delta(q))$.

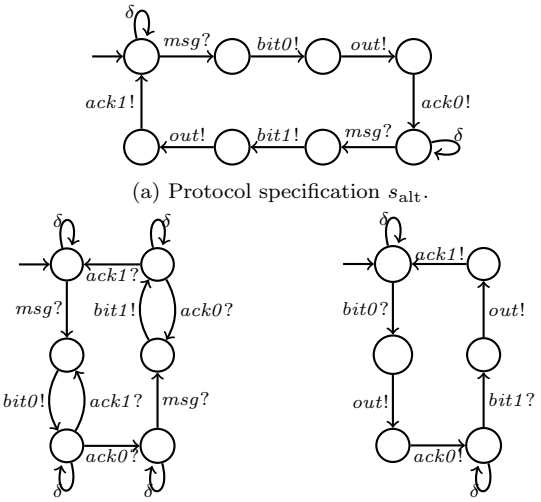
  There are no transitions for the inconsistent states.

Note that all states of the quotient reachable from the initial state are closed, i.e. $q = cl(q)$. Intuitively, the closure collects all additional requirements arising from the (uncontrollable) asynchronous moves of the known component.

Example 3. We illustrate the quotient operator on the alternating-bit protocol shown in Figure 8a, where $O_{s_{\text{alt}}} = \{out, bit0, bit1, ack0, ack1\}$, $I_{s_{\text{alt}}} = \{msg\}$. This protocol is realized by a transmitter and receiver modules, which exchange messages with a correction bit initially set to 0. The transmitter waits for an input message *msg?* and sends it to the receiver with a control bit (action *bit0!*). The receiver delivers the message *out!* and confirms to the transmitter by action *ack0!*. In the next round, the control bit is flipped and the protocol restarts.

Suppose we have already tested the transmitter against the specification c_{tr} in Figure 8a, where $O_{c_{\text{tr}}} = \{bit0, bit1\}$, $I_{c_{\text{tr}}} = \{msg, ack0, ack1\}$. The specification c_{tr} behaves like required by the protocol, but it has the additional feature that when it receives an out-of-order acknowledgment, it resends the bit notification. To obtain the specification of the receiver, we compute the quotient of s_{alt} by c_{tr} over the alphabet $I_{s_{\text{alt}}/c_{\text{tr}}} = \{bit0, bit1\}$. The resulting specification is shown in Figure 8c; for clarity we omitted the universal states.

Observe that the construction may create a suspension automaton that is not necessarily valid. To obtain a valid



(a) Protocol specification s_{alt}.

(b) Transmitter c_{tr}.

(c) The quotient of s_{alt} by c_{tr}.

Figure 8: The alternating-bit protocol.

suspension automaton we use the pruning operation presented in the previous section and several other operations announced in Fig. 5 and defined below. If the initial state s/c is removed in the pruning phase, we say that the quotient s/c does not exist.

Validization

We now show the procedure valid that converts any (generally invalid) suspension automaton $(Q, I, O \cup \{\delta\}, T, s)$ to one that is valid. This post-processing step yields the complete quotient operation. The algorithm proceeds in several phases:

- Pruning: In order to ensure the non-blocking property, we remove the blocking states, which may have been generated, similarly to the merging operation by applying the pruning procedure.
- δ-contraction: In order to ensure the quiescence-observation property, we apply the merge operation on the states of the single suspension automaton as follows. For $q \in Q$, let $\Delta(q) = \{q' \mid q \xrightarrow{\delta^*} q'\}$ denote all the δ-reachable states. We replace every transition $q \xrightarrow{\delta} q'$ by $q \xrightarrow{\delta} \bigwedge_{q' \in \Delta(q)} q'$ with a δ-self-loop on the latter state. Since the conjunction contains also q, it can only have less traces than q. Further δ-transitions do not change the state and thus preserve all the traces; thus the property is satisfied. Further, by the definition of **ioco**, for a quiescent i if $i \,\mathbf{ioco}\, q$ then also $i \,\mathbf{ioco}\, q'$ for each $q' \in \Delta(q)$, whence the preservation of implementations.
- Removing outputs after quiescence: In order to ensure the anomaly-freedom property, we need to prohibit outputs right after a quiescent transition was taken. Formally, we make a merge of s with the "universal system" described in Section 2.1. Intuitively, this duplicates the state space remembering whether the previous transition was quiescent or not, and thus prohibiting outputs or not, respectively.
- Pruning: Since the previous two merging phases may again create blocking states, we perform the pruning again.

As no other properties are violated by merging and none is violated by pruning, the final result satisfies all three properties and is a valid suspension automaton.

THEOREM 2. *Let s be a (possibly invalid) suspension automaton. Then* $\mathsf{valid}(s)$ *satisfies the axiom* **(V)**.

The quotient operator is given by the /-construction followed by application of valid.

THEOREM 3. *Let s, c be valid suspension automata. The quotient operator satisfies the axiom* **(Q)**. *In particular, the initial state is pruned if and only if there is no IOTS i such that $i \parallel j$* **ioco** *s for all j* **ioco** *c.*

While the merge in the third phase at most duplicates the state space, the merge of the second phase may in general cause an exponential blowup in connection with δ transitions. Fortunately, this is not the case when used as the quotient post-processing. During the quotient post-processing, instead of conjunction we can consider taking union and set $q \xrightarrow{\delta} \bigcup_{q' \in \Delta(q)} q'$. Indeed, in the quotient construction, each new state is a set of original states and implicitly represents their merge.

LEMMA 2. *Let q, q' (and thus also $q \cup q'$) be states of a quotient such that $q \wedge q'$ exists. Then for all IOTS i*

$$i \,\mathbf{ioco}\, q \wedge q' \iff i \,\mathbf{ioco}\, q \cup q'.$$

As a result, only singly exponential blow-up can be caused by the quotient operation.

3.3 Unhiding / Alphabet Equalization

In this section, we present the dual to hiding, the **unhide** operator. The purpose of this operator is to perform alphabet equalization, e.g. to change a partial specification view into a full specification. This becomes useful when combined with our previous operations: combining alphabet equalization with the merge operator allows us to merge partial specifications with different alphabets, while combining it with the quotient operator provides us with the decomposition in various architectural patterns.

It is common to only allow hiding output actions; however, we also permit hiding input actions in order to handle more general settings. Note that we treat input actions differently than outputs. While output actions are renamed into silent τ actions, input actions are simply removed.

Definition 5. Let (Q, I, O, T, s) be an IOLTS and let $I_H \subseteq I$, $O_H \subseteq O$ be sets of actions to be hidden. The IOLTS $\mathbf{hide}\ (I_H, O_H)\ \mathbf{in}\ s$ is $(Q, I \setminus I_H, O \setminus O_H, T', s)$ with transitions defined as follows:

- whenever $a \in (I \cup O \cup \{\tau\}) \setminus (I_H \cup O_H)$ and $s \xrightarrow{a} s'$ then $\mathbf{hide}\ (I_H, O_H)\ \mathbf{in}\ s \xrightarrow{a} \mathbf{hide}\ (I_H, O_H)\ \mathbf{in}\ s'$, and
- whenever $a \in O_H$ and $s \xrightarrow{a} s'$ then $\mathbf{hide}\ (I_H, O_H)\ \mathbf{in}\ s \xrightarrow{\tau} \mathbf{hide}\ (I_H, O_H)\ \mathbf{in}\ s'$.

We sometimes omit the set of input actions if it is empty, thus writing $\mathbf{hide}\, O_H\, \mathbf{in}\, s$ instead of $\mathbf{hide}\ (\emptyset, O_H)\ \mathbf{in}\ s$. This notation agrees with the standard **hide** operator as has been used in [16].

Ideally, the dual to hiding, the **unhide** operator would satisfy the following axiom: for all IOLTS i

$$\mathbf{hide}(I_H, O_H)\,\mathbf{in}\, i \,\mathbf{ioco}\ s \iff i \,\mathbf{ioco}\, \mathbf{unhide}(I_H, O_H)\,\mathbf{in}\, s$$

However, this is not possible in general. The problem lies in the fact that the implementation i may contain loops labeled with actions of O_H. After hiding, these loops introduce divergence, which means that the left-hand side of the equivalence is not well defined. We thus restrict ourselves to implementations that do not contain such loops; we call these implementations *hidden-output convergent*.

Definition 6. Let (Q, I, O, T, s) be an IOLTS and let I_H, O_H be disjoint sets of actions with $(I_H \cup O_H) \cap (I \cup O) = \emptyset$. The IOLTS $\mathbf{unhide}\ (I_H, O_H)\ \mathbf{in}\ s$ is defined as $(Q, I \cup I_H, O \cup O_H, T', s)$ where $T' = T \cup \{(q, a, q) \mid q \in Q, a \in O_H\}$.

Note that we have not explicitly defined any transitions for actions from I_H. Such actions (implicitly leading into the universal state) are going to be added once the IOLTS is changed into an input-enabled suspension automaton.

THEOREM 4. *Let (Q, I, O, T, s) be an IOLTS, I_H, O_H as in Definition 6. Then for all hidden-output convergent IOTS i the following holds:*

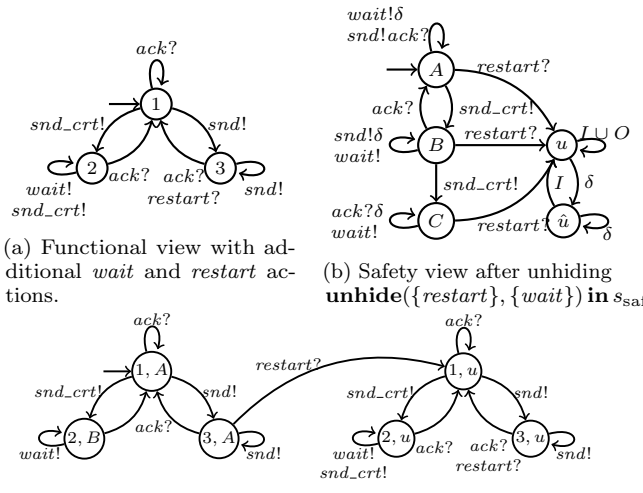
$$\mathbf{hide}(I_H, O_H)\,\mathbf{in}\, i \,\mathbf{ioco}\, s \iff i \,\mathbf{ioco}\, \mathbf{unhide}(I_H, O_H)\,\mathbf{in}\, s$$

The **unhide** operator allows us to extend the merge operator to systems with different alphabets, e.g. partial specifications. Let $s_1, \ldots, s_n$ be IOLTS (or their suspension automata) over input and output alphabets I_k, O_k, respectively, for each $k = 1, \ldots, n$. Further, let I and O be input and output alphabets that subsume all I_k and all O_k, respectively, and such that $I \cap O = \emptyset$. Then the *merge* $\bigwedge_{k=1}^{n} s_k$ *over alphabets* I, O is defined as the construction

$$\bigwedge_{k=1}^{n} \mathbf{unhide}(I \setminus I_k, O \setminus O_k)\,\mathbf{in}\, s_k$$

followed by pruning. As a corollary of Theorem 4, we thus obtain the main contribution of this section.

COROLLARY 1. *The merge operator over I, O satisfies the axiom* **(M)** *for all IOTS i that are hidden-output convergent with respect to all $O \setminus O_k$.*



(a) Functional view with additional *wait* and *restart* actions.

(b) Safety view after unhiding $\mathbf{unhide}(\{restart\}, \{wait\})\,\mathbf{in}\, s_{\text{safe}}$.

(c) Merge of the views in Figures 9a and 9b.

Figure 9: Example of merge with alphabet equalization.

Example 4. Figure 9a shows the functional view s_{fun} from Figure 6a extended by new actions *wait*! and *restart*?, and without δ-actions. The action *wait*! allows the dispatcher to become idle in state 2 and *restart*? resets the dispatcher from state 3 to the initial state. To equalize the alphabets, we unhide the safety view s_{safe} with the actions *restart*?, *wait*! as shown in Figure 9b; here the universal states are explicit. Finally, we merge the two views to obtain the specification in Figure 9c.

Similarly, we can consider parallel composition with the subsequent hiding and obtain the respective quotient operator, where **unhide** is applied on the specification of the whole system. In the following corollary we use the notation $hoc_{O_H}(i \parallel j)$ to denote that $i \parallel j$ is hidden-output convergent with respect to O_H.

COROLLARY 2. *Let s, c be specifications, $I_{s/c}$ be an input alphabet for the quotient, I_H and O_H be hiding alphabets. Then for all implementations i we have*

$$i \,\mathbf{ioco}(\mathbf{unhide}(I_H, O_H)\,\mathbf{in}\, s)/c \iff \forall j \,\mathbf{ioco}\, c : (hoc_{O_H}(i \parallel j) \Rightarrow \mathbf{hide}(I_H, O_H)\,\mathbf{in}(i \parallel j)\,\mathbf{ioco}\, s).$$

4. CONCLUSION AND FUTURE WORK

We proposed two operations coming from interface theories, merge and quotient, for exploiting structural properties of systems in **ioco**-based testing. Together with parallel composition, these operations enable minimizing the testing effort by: (1) facilitating multiple view modeling that follows the common structure of requirements documents; and (2) formalizing the notion of design patches. The operations are complete with respect to the standard semantics **(M)**, **(Q)**, in contrast to incompleteness of parallel composition **(PC)**. We ensured generality of the operations by allowing variable alphabets and flexible management of synchronization actions. Finally, the validization procedure is of independent interest in other settings [17].

We plan to investigate deeper the interplay between parallel composition, merging, and quotient in order to develop a methodology that further optimizes black-box testing of complex systems. We will also study the models with τ-divergent loops that are forbidden by the classical **ioco**-testing theory. We believe that such restriction is unnecessarily strong and not practical since such models may easily result from hiding of synchronization actions. Hence, we will study this problem and propose alternative solutions for handling models with τ-divergent loops.

Acknowledgments

This research was funded in part by the European Research Council (ERC) under grant agreement 267989 (QUAREM), by the Austrian Science Fund (FWF) projects S11402-N23 (RiSE) and Z211-N23 (Wittgestein Award), by People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme (FP7/2007-2013) under REA grant agreement 291734, and by the ARTEMIS JU under grant agreement 295373 (nSafeCer). Jan Křetínský has been partially supported by the Czech Science Foundation, grant No. P202/12/G061. Nikola Beneš has been supported by the MEYS project No. CZ.1.07/2.3.00/30.0009 Employment of Newly Graduated Doctors of Science for Scientific Excellence.

5. REFERENCES

[1] B. K. Aichernig, F. Lorber, D. Ničković, and S. Tiran. Require, test and trace it. Technical Report IST-MBT-2014-03, Graz University of Technology, 2014. `https://online.tugraz.at/tug_online/voe_main2.getVollText?pDocumentNr=637834&pCurrPk=77579`.

[2] S. Ben-David, M. Chechik, and S. Uchitel. Merging partial behaviour models with different vocabularies. In *CONCUR*, 2013.

[3] N. Beneš, I. Černá, and J. Křetínský. Modal transition systems: Composition and LTL model checking. In *ATVA*, 2011.

[4] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple viewpoint contract-based specification and design. In *FMCO*, 2007.

[5] P. Daca, T. A. Henzinger, W. Krenn, and D. Nickovic. Compositional specifications for ioco testing. In *ICST*, 2014.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, 2001.

[7] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In *EMSOFT*, 2001.

[8] L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT*, 2008.

[9] D. Fischbein, N. D'Ippolito, G. Brunet, M. Chechik, and S. Uchitel. Weak alphabet merging of partial behavior models. *ACM Trans. Softw. Eng. Methodol.*, 21(2):9, 2012.

[10] K. G. Larsen and L. Xinxin. Equation solving using modal transition systems. In *LICS*, 1990.

[11] N. Noroozi, M. R. Mousavi, and T. A. C. Willemse. On the complexity of input output conformance testing. In *FACS*, 2013.

[12] J. Raclet. Residual for component specifications. *Electr. Notes Theor. Comput. Sci.*, 215:93–110, 2008.

[13] J. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. A modal interface theory for component-based design. *Fundam. Inform.*, 108(1-2):119–149, 2011.

[14] J. Raclet, E. Badouel, A. Benveniste, B. Caillaud, and R. Passerone. Why are modalities good for interface theories? In *ACSD*, 2009.

[15] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[16] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *FATES*, 2003.

[17] T. A. C. Willemse. Heuristics for ioco-based test-based modelling. In *FMICS/PDMC*, 2006.

APPENDIX

To make some of the reasoning in the proofs easier, we first describe the notion of *coinductive* **ioco**. The coinductive **ioco** was defined in [11] where it is also shown that it coincides with standard **ioco** when the right-hand side specification is deterministic. Here, we present a slightly modified version. Instead of deterministic IOLTS on the right-hand side, we consider *possibly blocking suspension automata*, i.e. suspension automata that might contain blocking states.

Definition 7. Given an IOTS (Q_i, I, O, T_i, i) and a possibly blocking suspension automaton $(Q_s, I, O \cup \{\delta\}, T_s, s)$, a binary relation $R \subseteq Q_i \times Q_s$ is called a *coinductive ioco from i to s* when $(i, s) \in R$ and for every $(j, r) \in R$ it holds that:

1. (Input simulation) if $r \xrightarrow{a} r'$ for $a \in I$, then for all $j' \in j \text{ after } a$ we have $(j', r') \in R$,
2. (Output simulation) if $j \stackrel{a}{\Rightarrow} j'$ for $a \in O \cup \{\delta\}$, then $r \xrightarrow{a} r'$ and for all $j' \in j \text{ after } a$ we have $(j', r') \in R$.

We write $i \preceq s$, when there exists a coinductive **ioco** relation from i to s.

Note that we use $\xrightarrow{a}$ in the output simulation part, which means that we only consider the explicit δ transitions of the possibly blocking suspension automaton.

If the possibly blocking suspension automaton has no blocking states, i.e. it is a suspension automaton according to Definition 3, the above definition is equivalent to that of [11] and the following lemma holds:

LEMMA 3. *Let i be a state of an IOTS and s be a state of a suspension automaton. Then i **ioco** s if and only if $i \preceq s$.*

Correctness of Pruning

We first show that by pruning the possibly blocking suspension automaton, we get a suspension automaton that has the same implementations with respect to $\preceq$.

LEMMA 4. *Let s be a state of a possibly blocking suspension automaton. If s is an invalid state then there is no coinductive ioco relation containing (i, s) for any state i of an IOTS.*

PROOF. Let R be a coinductive ioco relation. The proof goes by induction. If s is a blocking state then any pair $(i, s) \in R$ would certainly violate the output simulation condition. If $s \xrightarrow{a?} s'$ where s' is an invalid state, then any pair $(i, s) \in R$ would require that for all $i' \in i \text{ after } a?$, $(i', s') \in R$. Note that $i \text{ after } a?$ is nonempty, as i is an IOTS state. However, by induction, (i', s') may not be in R as s' is an invalid state. The remaining case is that all output and δ transitions outgoing from s end in an invalid state. There is $a \in O \cup \{\delta\}$ such that $i \stackrel{a}{\Rightarrow} i'$. But then, either $s \stackrel{a}{\not\rightarrow}$, which violates the output simulation condition, or $s \xrightarrow{a} s'$ where s' is an invalid state. Again, by induction, the condition $(i', s') \in R$ cannot be satisfied. □

LEMMA 5. *Let s be a state of a possibly blocking suspension automaton. If s is not an invalid state then for every IOTS state i holds: $i \preceq s$ before pruning if and only if $i \preceq s$ after pruning.*

PROOF. Due to previous lemma, coinductive ioco relations are not affected by the pruning, as they can never contain pairs (j, t) with t an invalid state. □

The consequence of these results is that we can now freely use $\preceq$ in the following proofs.

Correctness of Merging

In this section, let $s_1, \ldots, s_n$ be a fixed input to the merge operation. Most notably, assume that all s_k have already been completed (Section 2.1) and thus are input enabled.

LEMMA 6. *Let i be an IOTS state satisfying $i \preceq s_k$ for all k. Then the merge $s = \bigwedge_{k=1}^{n} s_k$ exists and $i \preceq s$.*

PROOF. Let $R = \{(j, (t_1, \ldots, t_n)) \mid \forall k : j \preceq t_k\}$. We show that R is a coinductive ioco relation from i to s. Clearly, $(i, s) \in R$. Let now $(j, (t_1, \ldots, t_n)) \in R$.

Let $(t_1, \ldots, t_n) \xrightarrow{a?} (t'_1, \ldots, t'_n)$. This means that $t_k \xrightarrow{a?} t'_k$ for all k. Let now $j' \in j \text{ after } a?$ be arbitrary. From $j \preceq t_k$ we know that $j' \preceq t'_k$. Therefore, $(j', (t'_1, \ldots, t'_n)) \in R$.

Let $j \stackrel{a}{\Rightarrow} j'$ for $a \in O \cup \{\delta\}$. From $j \preceq t_k$ we know that $t_k \xrightarrow{a} t'_k$ and $j' \preceq t'_k$ for all k. Therefore, $(t_1, \ldots, t_n) \xrightarrow{a} (t'_1, \ldots, t'_n)$ and $(j', (t'_1, \ldots, t'_n)) \in R$.

We have shown that R is a coinductive ioco relation from i to s, therefore $i \preceq s$. □

LEMMA 7. *Let i be an IOTS state satisfying $i \preceq s$ with s as above. Let $k \in \{1, \ldots, n\}$ be arbitrary. Then $i \preceq s_k$.*

PROOF. Let $R = \{(j, t_k) \mid j \preceq (t_1, \ldots, t_n)\}$. We show that R is a coinductive ioco relation from i to s_k. Clearly, $(i, s_k) \in R$. Let now $(j, t_k) \in R$.

Let $t_k \xrightarrow{a?} t'_k$. Due to input enabledness and determinism, $(t_1, \ldots, t_n) \xrightarrow{a?} (t'_1, \ldots, t'_n)$. Let $j' \in j \text{ after } a?$ be arbitrary. From $j \preceq (t_1, \ldots, t_n)$ we know that $j' \preceq (t'_1, \ldots, t'_n)$. This means that $(j', t'_k) \in R$.

Let $j \stackrel{a}{\Rightarrow} j'$ for $a \in O \cup \{\delta\}$. From $j \preceq (t_1, \ldots, t_n)$ we know that $(t_1, \ldots, t_n) \xrightarrow{a} (t'_1, \ldots, t'_n)$ with $j' \preceq (t'_1, \ldots, t'_n)$. By the construction of merge, this means that $t_k \xrightarrow{a} t'_k$. Therefore, $(j', t'_k) \in R$.

We have shown that R is a coinductive ioco relation from i to s_k, therefore $i \preceq s_k$. □

Correctness of Quotient

We now focus on proving the correctness of the quotient construction. In this section, let s and c be fixed inputs for the construction. Denote $L_s = I_s \cup O_s$ and $L_c = I_c \cup O_c$ their respective alphabets and $L_{s/c} = I_{s/c} \cup O_{s/c}$ the alphabet of the quotient. Finally, for the sake of more readable notation, we write s/c instead of (s, c) for pairs of states of the system specification and the known component.

The following auxiliary lemma follows straightforwardly from the definition of coinductive ioco.

LEMMA 8. *If $i \preceq s$ and $i \stackrel{\varepsilon}{\Rightarrow} i'$, then $i' \preceq s$.*

We further observe the following two facts about the states of the quotient. The first is that if there exists an implementation $i \preceq q$ then q is consistent. The second is that all states of the quotient satisfy $q = cl(q)$. We call this second observation *the closure of q* in the following proofs.

LEMMA 9. *Let $\hat{q}$ be a state of the quotient and let $\hat{\imath} \preceq \hat{q}$. Let $s/c \in \hat{q}$ and let $\hat{\jmath} \preceq c$ be arbitrary. Then $\hat{\imath} \parallel \hat{\jmath} \preceq s$.*

PROOF. Let $R = \{(i \parallel j, t) \mid i \preceq q, t/d \in q, j \preceq d\}$. We show that R is a coinductive ioco relation. Let $(i \parallel j, t) \in R$.

Let $t \xrightarrow{a} t'$ for $a \in I_s$. This means that $a \in (I_c \setminus L_{s/c}) \cup (I_{s/c} \setminus L_c)$. Let further $i \parallel j \stackrel{a}{\Rightarrow} i' \parallel j'$ be arbitrary.

- Let $a \in I_c \setminus L_{s/c}$. Then $i \stackrel{\varepsilon}{\Rightarrow} i'$ and $j \stackrel{a}{\Rightarrow} j'$. This means that $i' \preceq q$ (previous lemma) and $j' \preceq d'$ such that $d \xrightarrow{a} d'$. Due to the closure of q, $t'/d' \in q$ and thus $(i' \parallel j', t') \in R$.
- Let $a \in I_{s/c} \setminus L_c$. Then $i \stackrel{a}{\Rightarrow} i'$ and $j \stackrel{\varepsilon}{\Rightarrow} j'$. This means that $i' \preceq q'$ such that $q \xrightarrow{a} q'$ and $j' \preceq d$. Due to the definition of $next_a(q)$, $t'/d \in q'$ and thus $(i' \parallel j', t') \in R$.

Let $i \parallel j \stackrel{a}{\Rightarrow} i' \parallel j'$ for $a \in O_c \cup O_{s/c} \cup \{\delta\}$.

- Let $a \in O_c \setminus L_{s/c}$. Then $i \stackrel{\varepsilon}{\Rightarrow} i'$ and $j \stackrel{a}{\Rightarrow} j'$. This means that $i' \preceq q$ and there exists $d \stackrel{a}{\rightarrow} d'$ with $j' \preceq d'$. If $t \not\stackrel{a}{\rightarrow}$ then q would be inconsistent; therefore, $t \stackrel{a}{\rightarrow} t'$ for some t'. Due to the closure of q, $t'/d' \in q$ and $(i' \parallel j', t') \in R$.
- Let $a \in O_{s/c} \setminus L_c$. Then $i \stackrel{a}{\Rightarrow} i'$ and $j \stackrel{\varepsilon}{\Rightarrow} j'$. This means that there has to exist $q \stackrel{a}{\rightarrow} q'$ with $i' \preceq q'$ and $j' \preceq d$. This means that $t \stackrel{a}{\rightarrow} t'$ for some t' and $t'/d \in q'$. Thus $(i' \parallel j', t') \in R$.
- Let $a \in L_{s/c} \cap L_c$. Then $i \stackrel{a}{\Rightarrow} i'$ and $j \stackrel{a}{\Rightarrow} j'$. This means that $q \stackrel{a}{\rightarrow} q'$ with $i' \preceq q'$ and $d \stackrel{a}{\rightarrow} d'$ with $j' \preceq d'$. This means that $t \stackrel{a}{\rightarrow} t'$ for some t' and $t'/d' \in q'$. Thus $(i' \parallel j', t') \in R$.
- Let $a = \delta$. Then $i \stackrel{\delta}{\Rightarrow} i'$ and $j \stackrel{\delta}{\Rightarrow} j'$. This means that there exists matching $q \stackrel{\delta}{\rightarrow} q'$ and $d \stackrel{\delta}{\rightarrow} d'$. Furthermore $q \stackrel{\delta}{\rightarrow}$ and $d \stackrel{\delta}{\rightarrow}$ imply $t \stackrel{\delta}{\rightarrow} t'$ for some t' and $t'/d' \in q'$. Thus $(i' \parallel j', t') \in R$.

We see that $(\hat{\imath} \parallel \hat{\jmath}, \hat{q}) \in R$ and therefore $\hat{\imath} \parallel \hat{\jmath} \preceq \hat{q}$. □

Before we prove the other direction, we need an auxiliary lemma.

LEMMA 10. *Let p be a suspension automaton state with $p \stackrel{\sigma}{\rightarrow} p'$ where σ is a trace that does not include δ and let $i' \preceq p'$. It is possible to construct an IOTS $i \preceq p$ such that $i \stackrel{\sigma}{\Rightarrow} i'$.*

PROOF. Let m be the length of σ and let $p = p_0 \stackrel{\sigma(1)}{\longrightarrow} p_1 \stackrel{\sigma(2)}{\longrightarrow} \cdots \stackrel{\sigma(m)}{\longrightarrow} p_m = p'$ be the abovementioned trace of p. We inductively construct i_k for k from m to 0 as follows:

- $i_m = i'$,
- for $k < m$, let i_k be an arbitrary implementation of p_k to which we add an extra transition $i_k \stackrel{\sigma(k+1)}{\longrightarrow} i_{k+1}$.

We then set $i = i_0$. It is obvious that i satisfies the statement of the lemma. □

LEMMA 11. *Let $\hat{q}$ be a state of the quotient and let $\hat{\imath}$ be an IOTS state such that for all $t/d \in \hat{q}$ and all $j \preceq d$, $\hat{\imath} \parallel j \preceq t$. Then $\hat{\imath} \preceq \hat{q}$.*

PROOF. Let $R = \{(i, q) \mid \forall t/d \in q, \forall j \preceq d, i \parallel j \preceq t\}$. We show that R is a coinductive ioco relation. Let $(i, q) \in R$.

Let $q \stackrel{a}{\rightarrow} q'$ for $a \in I_{s/c}$. Let further $i \stackrel{a}{\Rightarrow} i'$ be arbitrary and take an arbitrary $t'/d' \in q'$ and $j' \preceq d'$. We need to show that $i' \parallel j' \preceq t'$.

- Let $a \in I_{s/c} \setminus L_c$. Then $t \stackrel{a \cdot \sigma}{\longrightarrow} t'$ and $d \stackrel{\sigma}{\rightarrow} d'$ for some $\sigma \in (L_c \setminus L_{s/c})^*$. Using Lemma 10, we can construct $j \preceq d$ with $j \stackrel{\sigma}{\Rightarrow} j'$. Thus $i \parallel j \stackrel{a \cdot \sigma}{\Longrightarrow} i' \parallel j'$. As $i \parallel j \preceq t$, we have $i' \parallel j' \preceq t'$. Thus $(i', q') \in R$.
- Let $a \in I_{s/c} \cap O_c$. Similarly to the previous case, $t \stackrel{a \cdot \sigma}{\longrightarrow} t'$ and $d \stackrel{a \cdot \sigma}{\longrightarrow} d'$ for some $\sigma \in (L_c \setminus L_{s/c})^*$. Again, using Lemma 10, we have $j \preceq d$ with $j \stackrel{a \cdot \sigma}{\Longrightarrow} j'$. Thus $i \parallel j \stackrel{a \cdot \sigma}{\Longrightarrow} i' \parallel j'$. As $i \parallel j \preceq t$, we have $i' \parallel j' \preceq t'$. Thus $(i', q') \in R$.

Let $i \stackrel{a}{\Rightarrow} i'$ for some $a \in O_{s/c}$. Then for all $t/d \in q$ there is $t \stackrel{a}{\rightarrow}$, otherwise there would be some j such that $i \parallel j \not\preceq t$. This means that $q \stackrel{a}{\rightarrow} q'$. We then proceed as in the previous case.

Let $i \stackrel{\delta}{\Rightarrow} i'$. Either all $t/d \in q$ satisfy $d \stackrel{\delta}{\not\rightarrow}$, in which case $q \stackrel{\delta}{\rightarrow} \emptyset$ and clearly $(i', \emptyset) \in R$ for any i'. Otherwise, there is some $t \stackrel{\delta}{\rightarrow}$. The proof then proceeds again similarly to the first case (with $\delta \cdot \sigma$).

We see that $(\hat{\imath}, \hat{q}) \in R$ and therefore $\hat{\imath} \preceq \hat{q}$. □

Correctness of Validization

Validization preserves implementations with respect to $\preceq$.

LEMMA 12. *For any possibly blocking suspension automaton s and IOTS i we have*

$$i \preceq s \iff i \preceq \mathsf{valid}(s)$$

PROOF. Preservation of implementations by pruning has been already proved above. Merging with the universal automaton obviously preserves implementations, as the universal automaton admits any implementation and merging has been proved to be correct. The only part that has to be proved is that of the δ-contraction. We show that whenever $s_0 \stackrel{\delta}{\rightarrow} s_1 \stackrel{\delta}{\rightarrow} \cdots \stackrel{\delta}{\rightarrow} s_m$ and $i \preceq s_0$, then for all i' such that $i \stackrel{\delta}{\Rightarrow} i'$ and for all $k \in \{0, \ldots, m\}$ it holds that $i' \preceq s_k$.

We first note that if $i \stackrel{\delta}{\Rightarrow} i'$ then also $i \stackrel{\varepsilon}{\Rightarrow} i'$, and by Lemma 8 we obtain $i' \preceq s_0$.

We then note that $i' \stackrel{\delta}{\Rightarrow} i'$ and by repeated application of the output simulation part of Definition 7 we have $i' \preceq s_k$ for all k. □

Correctness of Unhiding

In the following, let I_H and O_H be fixed. We use the shorthand notation $h(i)$ for $\mathbf{hide}(I_H, O_H)\,\mathbf{in}\,i$ and $unh(s)$ for $\mathbf{unhide}(I_H, O_H)\,\mathbf{in}\,s$.

LEMMA 13. *Let i be a hidden-output bounded implementation such that $h(i) \preceq s$. Then $i \preceq unh(s)$.*

PROOF. We show that $R = \{(i, unh(s)) \mid h(i) \preceq s\} \cup \{(i, u), (i, \bar{u}) \mid i \text{ implementation }\}$ is a coinductive ioco relation. Clearly, the pairs (i, u) and $(i, \bar{u})$ satisfy the conditions of Definition 7. Let $(i, unh(s)) \in R$.

Let $unh(s) \stackrel{a}{\rightarrow} u$ where $a \in I_H$. Then for all $i' \in i\,\mathsf{after}\,a$, $(i', u) \in R$.

Let $unh(s) \stackrel{a}{\rightarrow} unh(s')$ where $a \in I_s$. Then $s \stackrel{a}{\rightarrow} s'$ and thus for all $h(i') \in h(i)\,\mathsf{after}\,a$, we have $h(i') \preceq s'$. Thus also $(i', unh(s')) \in R$.

Let $i \stackrel{a}{\Rightarrow} i'$ where $a \in O_H$. Then $unh(s) \stackrel{a}{\rightarrow} unh(s)$ and $h(i) \stackrel{\varepsilon}{\Rightarrow} h(i')$. By Lemma 8, we have $h(i') \preceq s$ and thus $(i', unh(s)) \in R$.

Let $i \stackrel{a}{\Rightarrow} i'$ where $a \in O_i \setminus O_H$. Then $h(i) \stackrel{a}{\Rightarrow} h(i')$ and due to $h(i) \preceq s$, we also have $s \stackrel{a}{\rightarrow} s'$. Thus also $unh(s) \stackrel{a}{\rightarrow} unh(s')$ and $(i', unh(s')) \in R$. □

LEMMA 14. *Let i be a hidden-output bounded implementation such that $i \preceq unh(s)$. Then $h(i) \preceq s$.*

PROOF. We show that $R = \{(h(i), s) \mid i \preceq unh(s)\}$ is a coinductive ioco relation. Let $(h(i), s) \in R$.

Let $s \stackrel{a}{\rightarrow} s'$. Then also $unh(s) \stackrel{a}{\rightarrow} unh(s')$ and thus for all $i' \in i\,\mathsf{after}\,a$, $i' \preceq unh(s')$. Therefore, $(h(i'), s') \in R$.

Let $h(i) \stackrel{a}{\Rightarrow} h(i')$. Then $i \stackrel{b_1}{\Rightarrow} i_1 \stackrel{b_2}{\Rightarrow} \cdots \stackrel{b_n}{\Rightarrow} i_n \stackrel{a}{\Rightarrow} i'$ where $b_k \in O_H$ for all k. Due to construction of unh and $i \preceq unh(s)$, we have $i_k \preceq unh(s)$ for all k. Due to $i_n \preceq unh(s)$, we have $unh(s) \stackrel{a}{\rightarrow} unh(s')$ with $i' \preceq unh(s')$. Therefore, $(h(i'), s') \in R$. □