CLEI electronic journal, Volume 25, Number 3, Paper 2 , November 2022

Conformance Checking and Pushdown Reactive Systems

Adilson Luiz Bonifacio
Computing Department, University of Londrina,
Londrina, Paraná, Brazil, 86051-990
bonifacio@uel.br

and

Arnaldo Vieira Moura
Computing Institute, University of Campinas,
Campinas, São Paulo, Brazil, 13083-970
arnaldo@ic.unicamp.br

Abstract

Due to their asynchronous interactions, testing reactive systems is a laborious activity present in any software development project. In this setting, the *finite memory* formalism of Labeled Transition Systems has been used to generate test suites that can be applied to check **ioco** conformance of implementations to a given specification. In this work we turn to a more complex scenario where a stronger formalism is considered, the Visibly Pushdown Labeled Transition System (VPTS), which allows access to a potentially *infinite pushdown memory*. We study an extension of the **ioco** conformance relation to VPTS models and develop polynomial time algorithms to verify conformance for VPTS models in a white-box testing scenario.

Keywords: Conformance checking, Test completeness, Reactive systems, Infinite memory.

1 Introduction

Reactive systems have become increasingly common among computer systems, whether used in simple technological solutions or in critical industrial applications. We see everywhere real-world systems being governed by reactive behaviors where the systems interact with an external environment by receiving input stimuli and producing outputs in response. Usually, the development of such systems requires precise and automatic support, especially in the testing activity, because high costs in terms of resources and maintenance time can be incurred when the test step is inappropriately performed.

Model-based testing is an important approach that has been employed to test reactive systems because it offers guarantees to the correctness of important properties, such as completeness of test suites, which can be formally proven [1, 2]. These aspects have been studied using appropriate models that capture the behavior of reactive systems, where the exchange of input and output stimuli can occur asynchronously. Prominent among such formalisms are Input/Output Labeled Transition Systems (IOLTSs) [3].

The process of testing is usually designed to verify whether an implementation is in compliance to a given specification [3]. This conformance checking process depends on the specification formalism, on the kind of fault model used, and on a conformance relation that is to be verified [3, 4]. For IOLTS models, Input/Output Conformance (**ioco**) is a well-studied relation [3, 5]. Recently, a more general approach [6], based on regular languages, has been proposed to check **ioco** conformance for IOLTS models.

In this work we mainly focus on pushdown reactive systems, more specifically, on a conformance checking framework for systems of this nature. We are not aware of any other approach that gives an efficient algorithm for checking conformance when treating models equipped with a pushdown memory. We study aspects of conformance testing and test suite generation for Input/Output Visibly Pushdown Labeled Transition (IOVPTS) models, inspired by the Visibly Pushdown Automata [7] formalism. Using an auxiliary pushdown stack, an IOVPTS can capture the behavior of much more complex reactive systems when compared to the simpler IOLTS models. Such is the case of an automatic vending machine that must keep track of the amount inserted, subtract the value of the purchase, and return the appropriate change. A proper model for such systems must use a formalism that allows for the manipulation of a potentially infinite memory. An illustration is discussed in Subsection 5.2 with the typical example of a drink dispensing machine, whose model requires a potentially infinite memory to formally describe its behavior as a reactive system. The same situation may also arise with other typical computational systems that are implemented via recursive programs. See Section 2 for comments on some related works that treat some aspects of recursive programs in the context of conformance testing.

Therefore we propose an extension of the **ioco** relation to IOVPTS models, in the sense that it prevents any observable implementation behavior that was not already present in the given specification. Our approach is based on a white-box testing scenario, where the structure of the implementation under test (IUT) is previously known by the tester. Specifications are assumed deterministic, but IUTSs can be even non-deterministic, and no further restrictions are required to constrain either on the specification or in the IUT models. The main novelty in this setting is to formally address the problem of developing algorithms of an acceptable complexity while still formally dealing with a potentially infinite pushdown memory.

Our algorithms always reach conclusive verdicts, either for conformance or for non-conformance, in polynomial time. When non-conformance is detected, a trace that witnesses the non-conformance is also generated. By contrast, in black-box scenarios, some restrictions must be assumed over the models and also limitations over the results. We prove the correctness of our algorithms and show that they run in worst case polynomial time in the size of both the specification and implementation.

We organize this paper as follows. Section 2 comments on some related works. In Section 3 we establish notations, define IOVPTSs and give preliminary results. Section 4 defines an ioco-like conformance relation for IOVPTS models and shows how to construct complete test suites. Section 5 develops an algorithm to test IUTs, proves its correctness, establishes a polynomial time bound for it, and illustrates these ideas on practical scenarios. Section 6 offers concluding remarks.

2 Related Works

Testing reactive systems has been largely studied in the literature, but several challenges still remain, mainly due to system and environment intrinsic complexities. Many researches of model-based testing usually deal with classical reactive systems, that is, they can be modeled by formalisms without memory, such as Labeled Transition Systems (LTSs) and their extension the Input/Output Labeled Transition Systems (IOLTSs).

In this work we study implementation testing in a scenario involving VPTSs, and their IOVPTS extensions, which are more complex and powerful models since they can access an unlimited pushdown memory. We develop a polynomial checking method that can be used to certify conformance between a given IOVPTS specification and a tentative IOVPTS implementation in a white-box testing scenario, where the structure of the implementation is known. Specifications models are assumed to be deterministic, but in the implementation side we can have non-deterministic models.

We now comment on some works more closely related to our approach. Constant et al. [8] propose a test case generation method for checking conformance of reactive systems in a black-box setting. Their method is based on algorithms that transform a set of recursive interprocedural specifications, with the semantics of the whole system being modeled as Push-Down Systems (PDSs). We note that the semantics of their formalism captures the so called Dyck languages, where each corresponding pair of symbols — the procedure calls and returns – appears properly balanced. These languages are a proper subset of the Visibly Pushdown Languages (VPLs), whose full extension is considered in this work. Their approach is confined to deterministic models and does not deal explicitly with arbitrary internal symbols as we do. Also quiescence is not treated, an important issue in a black-box scenario. Since they make use of test purposes, modeled by simple LTSs, their algorithms might lead to inconclusive verdicts when a state is reached from which it is impossible to reach an *accept* state in the test purpose model. In our scenario, conclusive verdicts are always emitted.

In more recent works, Dyck languages are also addressed in terms of the Dyck reachability problem which, in turn, is reduced to a reachability analysis when searching for a witness on checking conformance. Li et al. [9] propose a Dyck reachability analysis in time complexity $O(n^7)$ when considering two Dyck languages. Kjelstrøm and Pavlogiannis [10] have treated the interleaved Dyck reachability problem and their different variants. They have found an efficient algorithm for the interleaved Dyck reachability problem for two Dyck languages with a time complexity bounded by $O(n^3\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function [11]. However, as we noticed the class of Dyck languages is a proper subset of the VPLs, so confining their results in a more restrict group of applications when compared to our proposal.

Chédor et al. [12] propose a test case generation method where specifications are modeled as the so called

Recursive Tile Systems (RTS), whose semantics is captured by the traces of an IOLTS with infinite states. They proceed by applying successive transformations to the original RTS that models the specification, and the testing scenario treats implementations as black-boxes. At each step, properties of the resulting RTSs can then be established by examining the traces of the corresponding infinite IOLTSs. As the authors acknowledge, their approach can lead to test cases with an exponential state-space complexity, even if the original specification models are deterministic. In our method we generate, in polynomial time, test cases which provides complete fault coverage in a white-box scenario, and when specifications are deterministic. Since they also make use of test purposes as an aid to the certification process, their testing algorithms can lead to inconclusive verdicts when the testing process reaches where test purpose state marked *Accept* can no longer be reached.

Other approaches were designed using reachability analyses and model checking algorithms as proposed by Esparza et al. [13] and Finkel et al. [14]. In part, our approach is also inspired by some of these ideas, such as when checking for *balanced runs*, in Section 5. Some other researches also focus on the synchronization problem of Input-driven Pushdown Automata [15, 16, 17], where witness configurations must be found over an infinite state transition system. All these approaches lead to polynomial time algorithms as mentioned in the previous related works. However, we are not aware of any other approach that deal with the conformance checking problem as we do, and with a more efficient time complexity algorithm than we have proposed in our work.

In [18] the authors take a somewhat different direction. They show how to synthesize reactive systems, given traces of input/output symbols that represent the reactive behavior of the system specification.

3 Reactive Models with Infinite Memory

In this section we present the Visibly Pushdown Labeled Transition System (VPTS) and introduce its variation, the Input/Output VPTS. But first we establish some notation that will be useful.

3.1 Basic Notation

Let X and Y be sets. We indicate by $\mathcal{P}(X) = \{Z \mid Z \subseteq X\}$ the power set of X, and $X - Y = \{z \mid z \in X, z \notin Y\}$ indicates set difference. An alphabet is any non-empty set of symbols. Let A be an alphabet. A word over A is any finite sequence $\sigma = x_1 \dots x_n$ of symbols in A, that is, $n \geq 0$ and $x_i \in A$ for all $i = 1, 2, \dots, n$. When $n = 0$, σ is the empty sequence, also indicated by ε. The set of all finite sequences, or words, over A is denoted by $A^\star$, and the set of all nonempty finite words over A is indicated by A^+. When we write $x_1 x_2 \dots x_n \in A^\star$, it is implicitly assumed that $n \geq 0$ and that $x_i \in A$, $1 \leq i \leq n$, unless noted otherwise. The length of a word α over A is indicated by $|\alpha|$. Let $\sigma = \sigma_1 \dots \sigma_n$ and $\rho = \rho_1 \dots \rho_m$ be words over A. The concatenation of σ and ρ, indicated by $\sigma\rho$, is the word $\sigma_1 \dots \sigma_n \rho_1 \dots \rho_m$. A language G over A is any set $G \subseteq A^\star$. If $G_1, G_2 \subseteq A^\star$, their product is the language $G_1 G_2 = \{\sigma\rho \mid \sigma \in G_1, \rho \in G_2\}$. If $G \subseteq A^\star$, then its complement is $\overline{G} = A^\star - G$.

We will also need the notion of a morphism between alphabets. Let A, B be alphabets, a *homomorphism*, or just a *morphism*, from A to B is a function $h : A \to B^\star$. Such a morphism can be inductively extended to a function $\widehat{h} : A^\star \to B^\star$, thus:

$$\widehat{h}(\sigma) = \begin{cases} \varepsilon & \text{if } \sigma = \varepsilon \\ h(a)\widehat{h}(\rho) & \text{if } \sigma = a\rho \text{ with } a \in A. \end{cases}$$

We can further lift $\widehat{h}$ to a function $\widetilde{h} : \mathcal{P}(A^\star) \to \mathcal{P}(B^\star)$ by letting $\widetilde{h}(G) = \bigcup_{\sigma \in G} \widehat{h}(\sigma)$, when $G \subseteq A^\star$. We may write h instead of $\widehat{h}$, or of $\widetilde{h}$, when no confusion can arise. When $a \in A$, we let $h_a : A \to A - \{a\}$ be the morphism where $h_a(a) = \varepsilon$, and $h_a(x) = x$ when $x \neq a$. Hence, $h_a(\sigma)$ erases all occurrences of a in the word σ.

3.2 Visibly Pushdown Labeled Transition Systems

A Labeled Transition System (LTS) is a formalism convenient to express an asynchronous exchange of messages between participating entities, in the sense that outputs do not have to occur synchronously with inputs, but are generated as separated events. Any LTS, however, has only a finite memory, represented by its set of states. A Visibly Pushdown Labeled Transition System (VPTS), on the other hand, has a pushdown memory associated to it, and thus can make use of a potentially infinite memory. The next definition is inspired by the notion of a Visibly Pushdown Automaton [7].

Definition 1 *A* Visibly Pushdown Labeled Transition System *(VPTS) over an input alphabet L is a tuple $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T\rangle$, where:*

— *S is a finite set of* states *or* locations*;*

— *$S_{in} \subseteq S$ is the set of* initial states*;*

— *L is a set of* labels*, or* action symbols*,, partitioned thus $L = L_c \cup L_r \cup L_i$;*

— *There is a special symbol $\varsigma \notin L$, the* internal action symbol*;*

— *Γ is a set of* pushdown symbols*. There is a special symbol $\perp \notin \Gamma$, the* bottom-of-stack symbol*;*

— *$T = T_c \cup T_r \cup T_i$, where $T_c \subseteq S \times L_c \times \Gamma \times S$, $T_r \subseteq S \times L_r \times \Gamma \cup \{\perp\} \times S$, and $T_i \subseteq S \times (L_i \cup \{\varsigma\}) \times \{\sharp\} \times S$, where $\sharp \notin \Gamma \cup \{\perp\}$ is a place-holder symbol.*

The class of VPTSs over a set of labels L will be indicated by $\mathcal{VP}(L)$.

Let $t = (p, x, Z, q) \in T$. If $t \in T_c$ we say that it is a *push-transition*. Its intended meaning is that $\mathcal{S}$, in state $p \in S$ and reading the *push symbol* x, changes to state q and pushes Z onto the stack. When $t \in T_r$ we have a *pop-transition*, with the intended meaning that, in state $p \in S$, reading the *pop symbol* $x \in L_r$ and having Z as the topmost symbol in the stack, $\mathcal{S}$ pops Z from the the stack and changes to state q. Further, when the stack is reduced to the bottom of stack symbol, $\perp$, then a pop move can be taken, leaving the stack unchanged. We have a *simple-transition* when $t \in T_i$ and $x \in L_i$, and we have an *internal-transition* when $t \in T_i$ and $x = \varsigma$. The meaning of a simple-transition t is to change from state p to state q, while reading the *simple symbol* x and leaving the stack unchanged. An internal-transition also changes from state p to state q leaving the stack unchanged, but reads no symbol from the input. The following definition makes these notions precise.

Definition 2 *Let $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T\rangle \in \mathcal{VP}(L)$. A* configuration *of $\mathcal{S}$ is a pair $(p, \alpha) \in S \times (\Gamma^\star\{\perp\})$. When $p \in S_{in}$ and $\alpha = \perp$, (p, α) is an* initial configuration*. The set of all configurations of $\mathcal{S}$ is indicated by $\mathcal{C}_\mathcal{S}$. Let $(q, \alpha) \in \mathcal{C}_\mathcal{S}$, and let $\ell \in L \cup \{\varsigma\}$. Then we write $(p, \alpha) \xrightarrow{\ell} (q, \beta)$ if there is a transition $(p, \ell, Z, q) \in T$, and either:*

1. *$\ell \in L_c$, and $\beta = Z\alpha$;*
2. *$\ell \in L_r$, and either (i) $Z \neq \perp$ and $\alpha = Z\beta$, or (ii) $Z = \alpha = \beta = \perp$;*
3. *$\ell \in L_i \cup \{\varsigma\}$, $Z = \sharp$ and $\alpha = \beta$.*

We call $(p, \alpha) \xrightarrow{\ell} (q, \beta)$ an elementary move *of $\mathcal{S}$, and we say that (p, ℓ, Z, q) is the transition* used *in the move $(p, \alpha) \xrightarrow{\ell} (q, \beta)$.*

It is clear that after any elementary move $(p, \alpha) \xrightarrow{\ell} (q, \beta)$ we have $(q, \beta) \in \mathcal{C}_\mathcal{S}$, that is, (q, β) is also a configuration of $\mathcal{S}$. Hence, $\rightarrow$ induces a binary relation on $\mathcal{C}_\mathcal{S}$.

Remark 1 *In figures, a push-transition (s, x, Z, q) will be graphically represented by x/Z_+ next to the corresponding arc from s to q. Similarly, the label x/Z_- will indicate a pop-transition (s, x, Z, q). A simple- or internal-transition $(s, x, \sharp, q)$ will be indicated by the label x next to the corresponding arc.*

Example 1 *Figure 1 represents a VPTS $\mathcal{S}_1 = \langle S, S_{in}, L, \Gamma, T\rangle$ with $S = \{s_0, s_1\}$, $S_{in} = \{s_0\}$, $L_c = \{b\}$, $L_r = \{c, t\}$, $L_i = \{\}$, and $\Gamma = \{Z\}$. We have a push-transition (s_0, b, Z, s_0), the pop-transitions (s_0, c, Z, s_1), (s_0, t, Z, s_1), (s_1, c, Z, s_1), (s_1, t, Z, s_1), and an internal-transition $(s_1, \varsigma, \sharp, s_0)$.* □

Paths in VPTS models are just chains of elementary moves.

Definition 3 *Let $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T\rangle \in \mathcal{VP}(L)$ and let $(p, \alpha), (q, \beta) \in \mathcal{C}_\mathcal{S}$.*

1. *A word $\sigma = l_1, \ldots, l_n \in (L \cup \{\varsigma\})^\star$ is a* path *from (p, α) to (q, β) if there are configurations $(r_i, \alpha_i) \in \mathcal{C}_\mathcal{S}$, $0 \leq i \leq n$, such that $(r_{i-1}, \alpha_{i-1}) \xrightarrow{l_i} (r_i, \alpha_i)$, $1 \leq i \leq n$, with $(r_0, \alpha_0) = (p, \alpha)$ and $(r_n, \alpha_n) = (q, \beta)$.*
2. *Let $\sigma \in L^\star$. We say that σ is an* observable path *from (p, α) to (q, β) in $\mathcal{S}$ if there is a path μ from (p, α) to (q, β) in $\mathcal{S}$ such that $\sigma = h_\varsigma(\mu)$.*

In both cases we say that the path starts at (p, α) and ends at (q, β), and we say that the configuration (q, β) is reachable *from (p, α). We also say that (q, β) is* reachable in $\mathcal{S}$ *if it is reachable from an initial configuration of $\mathcal{S}$.*

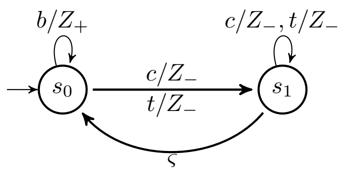


Figure 1: A VPTS $\mathcal{S}_1$.

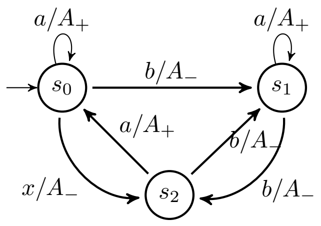


Figure 2: An IOVPTS specification $\mathcal{S}$.

Moves labeled by the internal symbol ς can occur in a path. An observable path is just a path from which ς-labels were removed. If σ is a path from (p, α) to (q, β), this can also be indicated by writing $(p, \alpha) \xrightarrow{\sigma} (q, \beta)$. When $|\sigma| = 1$ this has exactly the same meaning as indicated in Definition 2, so that no confusion can arise. We may also write $(p, \alpha) \xrightarrow{\sigma}$ to indicate that there is some $(q, \beta) \in \mathcal{C}_\mathcal{S}$ such that $(p, \alpha) \xrightarrow{\sigma} (q, \beta)$; likewise, $(p, \alpha) \rightarrow (q, \beta)$ means that there is some $\sigma \in (L \cup \{\varsigma\})^*$ such that $(p, \alpha) \xrightarrow{\sigma} (q, \beta)$. Also $(p, \alpha) \rightarrow$ means $(p, \alpha) \xrightarrow{\sigma} (q, \beta)$ for some $(q, \beta) \in \mathcal{C}_\mathcal{S}$ and some $\sigma \in (L \cup \{\varsigma\})^*$. When σ is an observable path from (p, α) to (q, β) we may write $(p, \alpha) \overset{\sigma}{\Rightarrow} (q, \beta)$, with similar shorthand notation also carrying over to the $\Rightarrow$ relation. When we want to emphasize that the underlying VPTS is $\mathcal{S}$, we write $(p, \alpha) \underset{\mathcal{S}}{\xrightarrow{\sigma}} (q, \beta)$, or $(p, \alpha) \underset{\mathcal{S}}{\overset{\sigma}{\Rightarrow}} (q, \beta)$.

Paths starting at (p, α) are also called the traces of (p, α), or the traces starting at (p, α). The semantics of a VPTS is related to traces starting at an initial configuration.

Definition 4 *Let $\mathcal{S} \in \mathcal{VP}(L)$ and let $(p, \alpha) \in \mathcal{C}_\mathcal{S}$.*

1. *The set of* traces *of (p, α) is $tr(p, \alpha) = \{\sigma \mid (p, \alpha) \xrightarrow{\sigma}\}$. The set of* observable traces *of (p, α) is $otr(p, \alpha) = \{\sigma \mid (p, \alpha) \overset{\sigma}{\Rightarrow}\}$.*
2. *The* semantics *of $\mathcal{S}$ is $tr(\mathcal{S}) = \bigcup_{q \in S_{in}} tr(q, \bot)$, and the* observable semantics *of $\mathcal{S}$ is $otr(\mathcal{S}) = \bigcup_{q \in S_{in}} otr(q, \bot)$.*

Clearly, $otr(\mathcal{S}) = h_\varsigma(tr(\mathcal{S}))$. If $\mathcal{S}$ has no internal transitions, then $otr(\mathcal{S}) = tr(\mathcal{S})$.

We can restrict the syntactic description of VPTS models somewhat, without loss of generality, by removing states that are not reachable from any initial state, since these states can not affect the system behavior. Moreover, we can also eliminate ς-labeled self-loops. We formalize these observations in the following hypothesis.

Hypothesis 1 *Let $\langle S, S_{in}, L, \Gamma, T\rangle \in \mathcal{VP}(L)$. For any $s \in S$ there is some $\alpha \in \Gamma^\star$ and $s_0 \in S_{in}$ such that $(s_0, \bot) \xrightarrow{\sigma} (s, \alpha\bot)$. Also, if $(s, \varsigma, \sharp, q) \in T$ then $s \neq q$.*

We see that a VPTS can autonomously move along ς-transitions, without consuming any input symbol. However, in some situations such moves may not be desirable, or simply we might want no observable behavior leading to two distinct states. This motivates the notion of determinism.

Definition 5 *Let $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T\rangle \in \mathcal{VP}(L)$. We say that $\mathcal{S}$ is* deterministic *if, for all s, $p \in S_{in}$, (s_1, β_1), $(s_2, \beta_2) \in \mathcal{C}_\mathcal{S}$, and all $\sigma \in L^\star$, we have that $(s, \bot) \overset{\sigma}{\Rightarrow} (s_1, \beta_1)$ and $(p, \bot) \overset{\sigma}{\Rightarrow} (s_2, \beta_2)$ imply $s_1 = s_2$ and $\beta_1 = \beta_2$.*

As a consequence, deterministic VPTSs have no internal moves.

Proposition 1 *If $\langle S, S_{in}, L, \Gamma, T\rangle \in \mathcal{VP}(L)$ is deterministic, then $\mathcal{S}$ has no ς-labeled transitions.*

Proof By contradiction, assume that $(s, \varsigma, \sharp, q) \in T$. From Remark 1 we get $s \neq q$ and we also get $\alpha \in \Gamma^\star$, $\sigma \in L^\star$ such that $(s_0, \bot) \xrightarrow{\sigma} (s, \alpha\bot)$, with $s_0 \in S_{in}$. Hence, $(s_0, \bot) \xrightarrow{\sigma} (s, \alpha\bot) \xrightarrow{\varsigma} (q, \alpha\bot)$. Using Definition 4 we get $(s_0, \bot) \overset{\mu}{\Rightarrow} (s, \alpha\bot)$ and $(s_0, \bot) \overset{\mu}{\Rightarrow} (q, \alpha\bot)$, where $\mu = h_\varsigma(\sigma)$. Since $s \neq q$, this contradicts Definition 5.

3.3 The Product of two VPTSs

It captures the synchronous behavior of the two models, and will be useful when testing conformance between two VPTS models.

Definition 6 *Let $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T\rangle$, $\mathcal{Q} = \langle Q, Q_{in}, L, \Delta, R\rangle \in \mathcal{VP}(L)$. Their product is the VPTS $\mathcal{S} \times \mathcal{Q} = \langle S \times Q, S_{in} \times Q_{in}, L, \Gamma \times \Delta, \nu\rangle$, where $((s_1, q_1), a, (Z_1, Z_2), (s_2, q_2)) \in \nu$ if either:*

1. *$a \in L_c \cup L_i$, $(s_1, a, Z_1, s_2) \in T$, $(q_1, a, Z_2, q_2) \in R$*

2. *$a \in L_r$, $(s_1, a, Z_1, s_2) \in T$, $(q_1, a, Z_2, q_2) \in R$ with either $Z_1 \neq \bot \neq Z_2$ or $Z_1 = Z_2 = \bot$*

3. *$a = \varsigma$, $Z = \sharp$, and either $s_1 = s_2$, $(q_1, \varsigma, \sharp, q_2) \in R$ or $q_1 = q_2$, $(s_1, \varsigma, \sharp, s_2) \in T$.*

The following result links moves in the original VPTSs to their product.

Proposition 2 *Let $\mathcal{S}, \mathcal{Q} \in \mathcal{VP}(L)$, and assume that*

$$(s, X_1 \cdots X_k\bot) \underset{\mathcal{S}}{\overset{\eta}{\rightarrow}} (r, W_1 \cdots W_n\bot) \text{ and } (q, Y_1 \cdots Y_k\bot) \underset{\mathcal{Q}}{\overset{\mu}{\rightarrow}} (t, Z_1 \cdots Z_m\bot),$$

where $k, n, m \geq 0$, and with $h_\varsigma(\eta) = h_\varsigma(\mu)$. Then, we have $n = m$ and

$$((s, q), U_1 \cdots U_k\bot) \underset{\mathcal{S}\times\mathcal{Q}}{\overset{\sigma}{\Rightarrow}} ((r, t), V_1 \cdots V_n\bot)$$

where $\sigma = h_\varsigma(\eta)$, $U_i = (X_i, Y_i)$ for $i = 1, \ldots, k$ and $V_i = (W_i, Z_i)$ for $i = 1, \ldots, n$.

Proof A routine induction on $|\eta| + |\mu| \geq 0$.

On the other direction, we have a similar result.

Proposition 3 *Let $\mathcal{S}, \mathcal{Q} \in \mathcal{VP}(L)$, and assume*

$$((s, q), U_1 \cdots U_k\bot) \underset{\mathcal{S}\times\mathcal{Q}}{\overset{\sigma}{\rightarrow}} ((r, t), V_1 \cdots V_n\bot)$$

with $U_i = (X_i, Y_i)$ for $i = 1, \ldots, k$, $V_i = (W_i, Z_i)$ for $i = 1, \ldots, n$, and $k, n \geq 0$. Then we have $(s, X_1 \cdots X_k\bot) \underset{\mathcal{S}}{\overset{\eta}{\rightarrow}} (r, W_1 \cdots W_n\bot)$, $(q, Y_1 \cdots Y_k\bot) \underset{\mathcal{Q}}{\overset{\mu}{\rightarrow}} (t, Z_1 \cdots Z_n\bot)$ with $h_\varsigma(\eta) = h_\varsigma(\sigma) = h_\varsigma(\mu)$.

Proof A simple induction on $|\sigma| \geq 0$.

3.4 Input Output Pushdown Transition Systems

The VPTS formalism can be used to model systems with a potentially infinite memory and with a capacity to interact asynchronously with an external environment. In such situations, we may want to treat some labels as symbols that the VPTS "receives" from the environment, and some other labels as symbols that the VPTS "sends back" to the environment. The next VPTS variation captures this idea.

Definition 7 *An Input/Output Visibly Pushdown Transition System (IOVPTS) is a tuple $\mathcal{I} = \langle S, S_{in}, L_I, L_U, \Gamma, T\rangle$, where*

- *L_I, L_U are finite sets of* input, output *labels, respectively, and $L_I \cap L_U = \emptyset$*
- *$\langle S, S_{in}, L_I \cup L_U, \Gamma, T\rangle$ is the* underlying VPTS *associated to $\mathcal{I}$.*

$\mathcal{IOVP}(L_I, L_U)$ is the class of all such IOVPTSs.

Remark 2 *In order to keep the number of definitions under control, we agree that in any reference to a notion based on IOVPTSs, and that has not been explicitly defined at some point, we substitute the IOVPTS model by its underlying VPTS.*

The semantics of an IOVPTS is just the set of its observable traces, that is, the observable traces of its underlying VPTS.

Definition 8 *Let $\mathcal{I} \in \mathcal{IOVP}(L_I, L_U)$. The* semantics *of $\mathcal{I}$ is the set $otr(\mathcal{I}) = otr(\mathcal{S}_\mathcal{I})$, where $\mathcal{S}_\mathcal{I}$ is the underlying VPTS associated to $\mathcal{I}$.*

Also, when referring an IOVPTS $\mathcal{I}$, the notation $\underset{\mathcal{I}}{\rightarrow}$ and $\underset{\mathcal{I}}{\Rightarrow}$ are to be understood as $\underset{\mathcal{S}}{\rightarrow}$ and $\underset{\mathcal{S}}{\Rightarrow}$, respectively, where $\mathcal{S}$ is the underlying VPTS associated to $\mathcal{I}$.

Example 2 *Figure 1 can be seen as an IOVPTS that describes a simple drink dispensing machine. In this case we have $L_I = \{b\}$ and $L_U = \{c, t\}$. From the context we can see that $L_c = \{b\}$, $L_r = \{c, t\}$ and $L_i = \emptyset$. The start state is s_0. Symbol b stands for button an user can press when asking for a cup of coffee or a cup of tea, with corresponding buttons represented by the labels c and t, respectively. Each time b button is activated, the model pushes the symbol Z on the stack, so that the stack is used to count how many times the b button was hit by the user.*

At any instant, after the user has activated the b button at least once, the machine moves to state s_1 and starts dispensing either coffee or tea, indicated by the c and t buttons. It decrements the stack each time a drink is dispensed, so that it will never deliver more drinks than the user asked for.

A move back to state s_0, over the internal label ς interrupts the delivery of drinks, so that the user can, possibly, receive less drinks than originally asked for. In this case, when the next user operates the machine it is possible, eventually, to collect more drinks than asked for. An alternative model could use one more state s_2 to interrupt the transition from s_1 to s_0 and install a self-loop at s_2 that empties the stack. A more realistic drink dispensing machine is illustrated in Subsection 5.2. □

We register one more example which will be used later.

Example 3 *Figure 2 depicts another IOVPTS, where $L_I = \{a, b\}$, $L_U = \{x\}$, $L_c = \{a\}$, $L_r = \{b, x\}$ and $L_i = \emptyset$. Also, $S_{in} = \{s_0\}$ and $\Gamma = \{A\}$.* □

4 Conformance Relation and Fault Models

In this section we provide a method to check whether IUTs, described as IOVPTSs, conform to a given specification IOVPTS model. We define an **ioco**-like conformance relation for IOVPTSs. The idea is that, given a specification $\mathcal{S}$ and an IUT $\mathcal{I}$, we say that $\mathcal{I}$ conforms to $\mathcal{S}$ when, for any observable behavior σ of $\mathcal{S}$, any output symbol that $\mathcal{I}$ can emit after running over σ is, necessarily, among the output symbols that $\mathcal{S}$ can also emit after it runs over the same σ. That notion captures the same behavior as the standard notion of **ioco**-conformance [3, 6] for LTS models, but the latter do not have access to an auxiliary pushdown store, as IOVPTSs models do.

4.1 An *ioco* Conformance Relation for IOVPTS Models

Let $\mathcal{S}$ be a specification and $\mathcal{I}$ an IUT. The **ioco**-like relation essentially says that if σ leads $\mathcal{I}$ to a configuration from which it can emit the output ℓ, then this must also hold for $\mathcal{S}$.

Definition 9 *Let $\mathcal{S} = \langle S, S_{in}, L_I, L_U, \Gamma, T\rangle$, $\mathcal{I} = \langle Q, Q_{in}, L_I, L_U, \Delta, R\rangle \in \mathcal{IOVP}(L_I, L_U)$, with $L = L_I \cup L_U$. Define*

1. *The function* **after**: $\mathcal{C}_\mathcal{S} \times L^\star \to \mathcal{P}(\mathcal{C}_\mathcal{S})$ *by letting*
$$(s, \alpha) \textbf{ after } \sigma = \big\{(q, \beta) \,|\, (s, \alpha) \overset{\sigma}{\Rightarrow} (q, \beta)\big\}, \text{ for all } (s, \alpha) \in \mathcal{C}_\mathcal{S}, \text{ and } \sigma \in L^\star.$$
2. *The function* **out**: $\mathcal{P}(\mathcal{C}_\mathcal{S}) \to L_U$ *thus*
$$\textbf{out}(V) = \bigcup_{(s,\alpha)\in V} \{\ell \in L_U \,|\, (s, \alpha) \overset{\ell}{\Rightarrow}\}.$$
3. $\mathcal{I}$ **ioco-like** $\mathcal{S}$ *if $\ell \in$ **out**$((q_0, \perp)$ **after** $\sigma)$ with $\sigma \in otr(\mathcal{S})$ and $q_0 \in Q_{in}$, then there is some $s_0 \in S_{in}$ such that $\ell \in$ **out**$((s_0, \perp)$ **after** $\sigma)$.*

Now we characterize the **ioco-like** conformance relation using the observable behaviors of both the specification and the given implementation.

Lemma 1 *Let $\mathcal{S}$, $\mathcal{I} \in \mathcal{IOVP}(L_I, L_U)$ and $D = otr(\mathcal{S})L_U$. Then, $\mathcal{I}$* **ioco-like** *$\mathcal{S}$ if and only if $otr(\mathcal{I}) \cap D \subseteq otr(\mathcal{S})$.*

Proof Write $\mathcal{S} = \langle S, S_{in}, L_I, L_U, \Gamma, T\rangle$ and $\mathcal{I} = \langle Q, Q_{in}, L_I, L_U, \Delta, R\rangle$.

Assume $otr(\mathcal{I}) \cap D \subseteq otr(\mathcal{S})$. Let $\sigma \in otr(\mathcal{S})$ and let $\ell \in$ **out**$((q_0, \perp)$ **after** $\sigma)$ for some $q_0 \in Q_{in}$. We must show that $\ell \in$ **out**$((s_0, \perp)$ **after** $\sigma)$ for some $s_0 \in S_{in}$. Because $\ell \in$ **out**$((q_0, \perp)$ **after** $\sigma)$ we get $\sigma, \sigma\ell \in otr(\mathcal{I})$. Since $\ell \in L_U$, we get $\sigma\ell \in otr(\mathcal{S})L_U$ and so $\sigma\ell \in D$. We conclude that $\sigma\ell \in otr(\mathcal{I}) \cap D$. Hence, $\sigma\ell \in otr(\mathcal{S})$, and so, $\ell \in$ **out**$((s_0, \perp)$ **after** $\sigma)$ for some $s_0 \in S_{in}$, as desired.

Next, assume that $\mathcal{I}$ **ioco-like** $\mathcal{S}$. Let $\sigma \in otr(\mathcal{I}) \cap D$. Then, $\sigma \in D$ and so $\sigma = \alpha\ell$ with $\ell \in L_U$ and $\alpha \in otr(\mathcal{S})$. Also, $\sigma \in otr(\mathcal{I})$ gives $\alpha\ell \in otr(\mathcal{I})$, and so $\alpha \in otr(\mathcal{I})$. Then, because $\ell \in L_U$, we get $\ell \in$ **out**$((q_0, \perp)$ **after** $\alpha)$ for some $q_0 \in Q_{in}$. Because we assumed $\mathcal{I}$ **ioco-like** $\mathcal{S}$ and we have $\alpha \in otr(\mathcal{S})$, we also get $\ell \in$ **out**$((s_0, \perp)$ **after** $\alpha)$, for some $s_0 \in S_{in}$. So $\alpha\ell \in otr(\mathcal{S})$. Because $\sigma = \alpha\ell$, we have $\sigma \in otr(\mathcal{S})$.

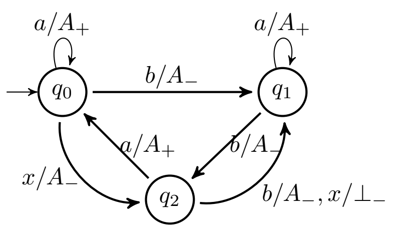


Figure 3: An IUT $\mathcal{I}$ with $L_I = \{a, b\}$ and $L_U = \{x\}$.

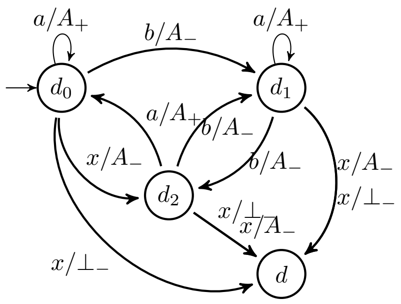


Figure 4: A VPTS for $\overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$ for the IOVPTS $\mathcal{S}$ of Example 2.

Example 4 *We illustrate Lemma 1, using the specification IOVPTS $\mathcal{S}$ depicted in Figure 2 and the implementation $\mathcal{I}$ depicted in Figure 3.*

*We want to check whether $\mathcal{I}$ **ioco-like** $\mathcal{S}$ holds. Let $\sigma = aabb$. From Figure 2 it is apparent that $(s_0, \bot) \overset{\sigma}{\Rightarrow} (s_2, \bot)$ and that $(s_0, \bot)$ **after** $\sigma = \{(s_2, \bot)\}$. From Figure 3 we get $(q_0, \bot)$ **after** $\sigma = \{(q_2, \bot)\}$. Also, $x \in$ **out**$((q_2, \bot))$, but $x \notin$ **out**$((s_2, \bot))$. So, by Definition 9, $\mathcal{I}$ **ioco-like** $\mathcal{S}$ does not hold.*

Now take $\sigma = aabbx$. Since $aabb \in otr(\mathcal{S})$, we get $aabbx \in otr(\mathcal{S})L_U = D$. Also, $aabbx \in otr(\mathcal{I})$ and $aabbx \notin otr(\mathcal{S})$, so that $aabbx \in otr(\mathcal{I}) \cap D \not\subseteq otr(\mathcal{S})$. □

We can also characterize the **ioco-like** relation as follows.

Corollary 1 *Let $\mathcal{S}$, $\mathcal{I} \in \mathcal{IOVP}(L_I, L_U)$. Then $\mathcal{I}$ **ioco-like** $\mathcal{S}$ if and only if $otr(\mathcal{I}) \cap T = \emptyset$, where $T = \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$.*

Proof Immediate from Lemma 1.

Example 5 *Let the IOVPTS $\mathcal{S}$ of Figure 2 be the specification, with $L_I = \{a, b\}$, and $L_U = \{x\}$. We want to construct a model for a test suite $T \subseteq (L_i \cup L_U)^\star$ that can be used for testing whether $\mathcal{I}$ **ioco-like** $\mathcal{S}$, for any implementation $\mathcal{I}$. From Corollary 1, we know that $\mathcal{I}$ **ioco-like** $\mathcal{S}$ is equivalent to $otr(\mathcal{I} \cap T) = \emptyset$, where $T = \overline{otr}(\mathcal{S}) \cap D$, where $D = otr(\mathcal{S})L_U$. So, we want a model that describes such a T.*

We start with a representation for $D = otr(\mathcal{S})L_U$. Consider the VPTS $\mathcal{TS}$ depicted in Figure 4. Note that x is the only symbol in L_U. $\mathcal{TS}$ was obtained from $\mathcal{S}$ by adding all possible transitions over x to the new state d of $\mathcal{TS}$, starting at all states of $\mathcal{S}$ from which that transition was not present in $\mathcal{S}$. Hence, it is clear that $D = otr(\mathcal{S})L_U$ is the set of all σ that lead $\mathcal{TS}$ from the initial configuration $(d_0, \bot)$ to a configuration $(d, \alpha\bot)$. Also, note that all transitions into state d_0 add a symbol A to the stack. Hence, $\mathcal{S}$ can never make a move on the pop symbol x from state s_0 when the stack is empty. The other possible moves of $\mathcal{TS}$ into d from states d_1 and d_2 obviously are not possible moves in $\mathcal{S}$. We conclude that all elements in D are also in $\overline{otr}(\mathcal{S})$. This gives $T = \overline{otr}(\mathcal{S}) \cap D = D$.

*Thus, to verify if an IUT $\mathcal{I}$ **ioco-like**-conforms to $\mathcal{S}$, we must check if there is some element in D that is also in the observable semantic of $\mathcal{I}$. In the next two subsections we use the notion of a fault model to make this procedure more systematic.* □

4.2 IOVPTS Fault Models

First, we formally model the external environment as an IOVPTS with a special **fail** state. Such a model, $\mathcal{T}$, operates in conjunction with an IUT, $\mathcal{I}$. Their joint behavior can be interpreted as $\mathcal{T}$ sending symbols to $\mathcal{I}$, and $\mathcal{I}$ responding by sending symbols back to $\mathcal{T}$. Note that, in this setting, the sets of input and output symbols in $\mathcal{T}$ and $\mathcal{I}$ must be interchanged.

Definition 10 *Let L_I and L_U be sets of input and output symbols, respectively. An* Input/Output Visibly Pushdown Fault Model *(*IOVPFM*) is any $\mathcal{T} \in \mathcal{IOVP}(L_U, L_I)$ with a distinguished* **fail** *state.*

Given an IOVPFM $\mathcal{T}$ and an IUT $\mathcal{I}$, the exchange of action symbols between $\mathcal{T}$ and $\mathcal{I}$ can be described by the product of their underlying VPTSs. Recall Definition 6.

Definition 11 *Let $\mathcal{S}$, $\mathcal{I} \in \mathcal{IOVP}(L_I, L_U)$ and $L = L_I \cup L_U$. Their product is $\mathcal{V}_\mathcal{S} \times \mathcal{V}_\mathcal{I}$, with $\mathcal{V}_\mathcal{S}$, $\mathcal{V}_\mathcal{I} \in \mathcal{VP}(L)$ being the underlying VPTSs of $\mathcal{S}$ and $\mathcal{I}$, respectively.*

In order to facilitate the notation, we will also denote the cross-product of $\mathcal{S}$ and $\mathcal{I}$ simply by $\mathcal{S} \times \mathcal{I}$. Having an implementation $\mathcal{I}$ and an IOVPFM $\mathcal{T}$ as a tester, we need to say when a test run is successful with respect to a given specification $\mathcal{S}$. Recalling that $\mathcal{T}$ signals an unsuccessful run when it reaches a **fail** state, we will say that an IUT $\mathcal{I}$ passes $\mathcal{T}$ when no synchronous execution of $\mathcal{T}$ and $\mathcal{I}$ reaches a $(\textsf{fail}, q)$ state in $\mathcal{T} \times \mathcal{I}$. In this case, we want $\mathcal{I}$ **ioco-like** $\mathcal{S}$ to hold. Alternatively, if some run of $\mathcal{T} \times \mathcal{I}$ does reach a $(\textsf{fail}, q)$ state, that is, if $\mathcal{I}$ does not pass $\mathcal{T}$, then we want a guarantee that $\mathcal{I}$ **ioco-like** $\mathcal{S}$ does not hold. In other words, we need a property of completeness.

Definition 12 *Let $\mathcal{T} = \langle Q, Q_{in}, L_U, L_I, \Delta, R\rangle \in \mathcal{IOVP}(L_U, L_I)$, and $\mathcal{I} = \langle S, S_{in}, L_I, L_U, \Gamma, T\rangle \in \mathcal{IOVP}(L_I, L_U)$. We say that $\mathcal{I}$* passes *$\mathcal{T}$ if, for all $\sigma \in (L_I \cup L_U)^\star$ and all initial configurations $((t_0, q_0), \perp)$ of $\mathcal{T} \times \mathcal{I}$ we do not have $((t_0, q_0), \perp) \underset{\mathcal{T}\times\mathcal{I}}{\overset{\sigma}{\Rightarrow}} ((\textsf{fail}, q), \alpha\perp)$, for any configuration $((\textsf{fail}, q), \alpha\perp)$ of $\mathcal{T} \times \mathcal{I}$. Also let $\mathcal{S} \in \mathcal{IOVP}(L_I, L_U)$. We say that $\mathcal{T}$ is* **ioco-like** complete *for $\mathcal{S}$ if we have $\mathcal{I}$* **ioco-like** *$\mathcal{S}$ if and only if $\mathcal{I}$ passes $\mathcal{T}$, for all $\mathcal{I} \in \mathcal{IOVP}(L_I, L_U)$.*

Now we construct an IOVPFM which is complete for a given specification $\mathcal{S}$.

Lemma 2 *Let $\mathcal{S} \in \mathcal{IOVP}(L_I, L_U)$ be deterministic with n states. We can effectively construct an IOVPFM $\mathcal{T}$ which is* **ioco-like** *complete for $\mathcal{S}$. Moreover, $\mathcal{T}$ is deterministic and has $n + 1$ states.*

Proof According to Definition 12, in order for $\mathcal{T}$ to be **ioco-like** complete for $\mathcal{S}$ we need that, for all implementations $\mathcal{I}$, it holds that $\mathcal{I}$ passes $\mathcal{T}$ if and only if $\mathcal{I}$ **ioco-like** $\mathcal{S}$. From Corollary 1 we know that $\mathcal{I}$ **ioco-like** $\mathcal{S}$ if and only if $otr(\mathcal{I}) \cap T = \emptyset$, where $T = \overline{otr}(\mathcal{S}) \cap [otr(\mathcal{S})L_U]$. That is, we need $\mathcal{T}$ such that $\mathcal{I}$ passes $\mathcal{T}$ if and only if $otr(\mathcal{I}) \cap T = \emptyset$.

Let $\mathcal{S} = \langle S_\mathcal{S}, S_{in}, L_I, L_U, \Delta_\mathcal{S}, T_\mathcal{S}\rangle$ be the given deterministic specification, $L = L_I \cup L_U$ and $n = |S_\mathcal{S}|$. The desired fault model $\mathcal{T} = \langle S_\mathcal{T}, T_{in}, L_U, L_I, \Delta_\mathcal{T}, T_\mathcal{T}\rangle$ is constructed as follows. Let $T_{in} = S_{in}$, $\Delta_\mathcal{T} = \Delta_\mathcal{S}$, and extend the state set $S_\mathcal{S}$ and the transition set $T_\mathcal{S}$ as follows. Define $S_\mathcal{T} = S_\mathcal{S} \cup \{\textsf{fail}\}$ where $\textsf{fail} \notin S_\mathcal{S}$. Fix some symbol $Z \in \Delta_\mathcal{S}$, and let

$$\begin{aligned} T_\mathcal{T} &= T_\mathcal{S} \\ &\cup \big\{(s, \ell, Z, \textsf{fail}) \,|\, \ell \in L_U \cap L_c \text{ and } (s, \ell, W, p) \notin T_\mathcal{S}, \text{ for any } p \in S_\mathcal{S}, W \in \Delta_\mathcal{S}\big\} && (1)\\ &\cup \big\{(s, \ell, W, \textsf{fail}) \,|\, \ell \in L_U \cap L_r \text{ and } (s, \ell, W, p) \notin T_\mathcal{S}, \text{for any } p \in S_\mathcal{S}\big\} && (2)\\ &\cup \big\{(s, \ell, \sharp, \textsf{fail}) \,|\, \ell \in L_U \cap L_i \text{ and } (s, \ell, \sharp, p) \notin T_\mathcal{S}, \text{ for any } p \in S_\mathcal{S}\big\} && (3) \end{aligned}$$

Since $\mathcal{S}$ has n states, it is clear that $\mathcal{T}$ has $n+1$ states and a single **fail** state. Further, using Proposition 1, we see that $\mathcal{S}$ has no ς-moves. Hence, by construction $\mathcal{T}$ has no ς-moves.

Claim 1. $\mathcal{T}$ is deterministic.

Proof. Let $\sigma \in (L_U \cup L_I)^\star$ and $(s_i, \perp) \underset{\mathcal{T}}{\overset{\sigma}{\Rightarrow}} (p_i, \alpha_i\perp)$ with $s_1, s_2 \in \mathcal{T}_{in} = \mathcal{S}_{in}$, $i = 1, 2$. According to Definition 5 we want to show that $s_1 = s_2$ and $\alpha_1 = \alpha_2$. Since $\mathcal{T}$ has no ς-moves, we can write $(s_i, \perp) \underset{\mathcal{T}}{\overset{\sigma}{\to}} (p_i, \alpha_i\perp)$, $i = 1, 2$.

If $p_1 \neq \textsf{fail} \neq p_2$ then, since **fail** is a sink state, we must have $(s_i, \perp) \underset{\mathcal{S}}{\overset{\sigma}{\to}} (p_i, \alpha_i\perp)$, $i = 1, 2$. The determinism of $\mathcal{S}$ gives the desired result in this case.

Now assume $p_1 \neq \textsf{fail}$, $p_2 = \textsf{fail}$. Since $s_1 \in S_{in}$ we get $s_1 \neq p_1$ so that $\sigma = \mu x$ with $x \in L_I \cup L_U$, and we can write $(s_i, \perp) \underset{\mathcal{T}}{\overset{\mu}{\to}} (r_i, \beta_i\perp) \underset{\mathcal{T}}{\overset{x}{\to}} (p_i, \alpha_i\perp)$, $i = 1, 2$. Again, we have $(s_i, \perp) \underset{\mathcal{S}}{\overset{\mu}{\to}} (r_i, \beta_i\perp)$ for $i = 1, 2$. The determinism of $\mathcal{S}$ implies $r_1 = r_2$ and $\beta_1 = \beta_2$. So, we have $(r_1, x, Z_1, \textsf{fail})$ in $\mathcal{T}$ and $(r_2, x, Z_2, p_2) = (r_1, x, Z_2, p_2)$ in $\mathcal{T}$, for some $Z_1, Z_2 \in \Delta_\mathcal{T}$. Clearly, (r_1, x, Z_2, p_2) is $\mathcal{S}$. If x is a push symbol, then $Z_1 = Z$. Now, (r_1, x, Z_2, p_2) in $\mathcal{S}$ and $(r_1, x, Z, \textsf{fail})$ in $\mathcal{T}$ contradict Eq. (1). Similarly when x is a simple symbol we get a contradiction to Eq. (3). When x is a pop symbol, let W be the first symbol in $\beta_1\perp = \beta_2\perp$. We now have $Z_1 = W = Z_2$ and then (r_1, x, W, p_2) in $\mathcal{S}$ and $(r_1, x, W, \textsf{fail})$ in $\mathcal{T}$ contradict to Eq. (2).

Lastly take $p_1 = \textsf{fail} = p_2$. This gives $\sigma = \mu x$ with $x \in L_I \cup L_U$ and $(s_i, \perp) \underset{\mathcal{T}}{\overset{\mu}{\to}} (r_i, \beta_i\perp) \underset{\mathcal{T}}{\overset{x}{\to}} (\textsf{fail}, \alpha_i\perp)$ for $i = 1, 2$. Again, $(s_i, \perp) \underset{\mathcal{S}}{\overset{\mu}{\to}} (r_i, \beta_i\perp)$ for $i = 1, 2$. The determinism of $\mathcal{S}$ implies $r_1 = r_2$ and $\beta_1 = \beta_2$. We now have transitions $(r_i, x, Z_i, \textsf{fail})$ in $\mathcal{T}$, $i = 1, 2$. If x is a push symbol, Eq. (1) gives $Z_1 = Z = Z_2$, so that $\alpha_1 = Z\beta_1 = Z\beta_2 = \alpha_2$ and we have the result. When x is a simple symbol, we get $\alpha_1 = \beta_1 = \beta_2 = \alpha_2$. Assume that x is a pop symbol. If $\beta_1 = \beta_2 \neq \varepsilon$ we must have $\beta_1 = \beta_2 = W\gamma$, and then $\alpha_1 = \gamma = \alpha_2$. Otherwise, we have $\beta_1 = \varepsilon = \beta_2$ and again $\alpha_1 = \perp = \alpha_2$.

We conclude that $\mathcal{T}$ is deterministic.

The next claim shows that any $\sigma \in \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$ leads $\mathcal{T}$ to the **fail** state.

Claim 2. Let $\sigma \in \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$. Then, $(s_0, \bot) \overset{\sigma}{\underset{\mathcal{T}}{\Rightarrow}} (\textbf{fail}, \alpha\bot)$ for some $s_0 \in T_{in}$, $\alpha \in (\Delta_{\mathcal{T}})^\star$.

Proof. Let $\sigma = \mu\ell$, $\sigma \notin otr(\mathcal{S})$, $\ell \in L_U$ and $\mu \in otr(\mathcal{S})$. Then, from Definition 3 and since $\mathcal{S}$ has no ς-moves, we get $(s_0, \bot) \overset{\mu}{\underset{\mathcal{S}}{\to}} (p, \alpha\bot)$, where $s_0 \in S_{in} = T_{in}$, $\alpha \in (\Delta_{\mathcal{S}})^\star$. By construction, all transitions in $\mathcal{S}$ are also transitions of $\mathcal{T}$, so that $(s_0, \bot) \overset{\mu}{\underset{\mathcal{T}}{\to}} (p, \alpha\bot)$.

We now argue that $(p, \alpha\bot) \overset{\ell}{\underset{\mathcal{T}}{\to}} (\textbf{fail}, \beta\bot)$ where $\beta \in (\Delta_{\mathcal{T}})^\star$. Composing we get $(s_0, \bot) \overset{\mu\ell}{\underset{\mathcal{T}}{\to}} (\textbf{fail}, \beta\bot)$, as needed. We note that we cannot have $(p, \alpha\bot) \overset{\ell}{\underset{\mathcal{S}}{\to}} (z, \gamma\bot)$ for any $z \in S_{\mathcal{S}}$, $\gamma \in (\Delta_{\mathcal{S}})^\star$, because then we would get $(s_0, \bot) \overset{\mu\ell}{\underset{\mathcal{S}}{\to}} (z, \gamma\bot)$, and then $\mu\ell = \sigma \in otr(\mathcal{S})$, a contradiction. There are three simple cases. If $\ell \in L_c$, then $(p, \ell, W, z) \notin T_{\mathcal{S}}$ for any $z \in S_{\mathcal{S}}$ and any $W \in \Delta_{\mathcal{S}}$. Then, Eq. (1) gives $(p, \ell, Z, \textbf{fail}) \in T_{\mathcal{T}}$, as needed. If $\ell \in L_i$, the reasoning is the same, using Eq. (3). Now let $\ell \in L_r$. Since $(p, \alpha\bot) \overset{\ell}{\underset{\mathcal{S}}{\to}} (z, \gamma\bot)$ is not allowed, we cannot have (p, ℓ, W, z) in $T_{\mathcal{S}}$ for any $z \in S_{\mathcal{S}}$, where $W \in \Delta_{\mathcal{S}}$ is the first symbol in $\alpha\bot$. Now, Eq. (2) gives $(p, \ell, W, \textbf{fail}) \in T_{\mathcal{T}}$. Since $\sigma = \mu\ell$ and $(s_0, \bot) \overset{\mu\ell}{\underset{\mathcal{T}}{\to}} (\textbf{fail}, \beta\bot)$, we get $(s_0, \bot) \overset{\sigma}{\underset{\mathcal{T}}{\Rightarrow}} (\textbf{fail}, \beta\bot)$.

The next claim deals with the converse.

Claim 3. Let $\sigma \in (L_I \cup L_U)^\star$, $(t_0, \bot) \overset{\sigma}{\underset{\mathcal{T}}{\Rightarrow}} (\textbf{fail}, \beta\bot)$ with $t_0 \in T_{in}$, $\beta \in (\Delta_{\mathcal{T}})^\star$. Then $\sigma \in \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$.

Proof. Since there are no ς-moves in $\mathcal{T}$, we must have $(t_0, \bot) \overset{\mu}{\underset{\mathcal{T}}{\to}} (p, \alpha\bot) \overset{\ell}{\underset{\mathcal{T}}{\to}} (\textbf{fail}, \beta\bot)$, with $\sigma = \mu\ell$.

Since **fail** is a sink state we get $p \neq \textbf{fail}$, and we know that all transitions in $(t_0, \bot) \overset{\mu}{\underset{\mathcal{T}}{\to}} (p, \alpha\bot)$ are in $\mathcal{S}$, so that $(t_0, \bot) \overset{\mu}{\underset{\mathcal{S}}{\to}} (p, \alpha\bot)$. Thus, $\mu \in otr(\mathcal{S})$. We must also have a transition $(p, \ell, X, \textbf{fail})$ in $\mathcal{T}$. By the construction, it can only be inserted in $T_{\mathcal{T}}$ by force of Equations (1),(2), or (2). In any case, we get $\ell \in L_U$. Hence $\sigma = \mu\ell \in otr(\mathcal{S})L_U$.

Assume that $\sigma \in otr(\mathcal{S})$. Since $\mathcal{S}$ has no ς-moves we get $(s_0, \bot) \overset{\mu}{\underset{\mathcal{S}}{\to}} (p', \alpha'\bot) \overset{\ell}{\underset{\mathcal{S}}{\to}} (r, \gamma\bot)$, with $s_0 \in S_{in}$, $p', r \in S_{\mathcal{S}}$, $\alpha', \gamma \in (\Delta_{\mathcal{S}})^\star$. We now have $(t_0, \bot) \overset{\mu}{\underset{\mathcal{S}}{\to}} (p, \alpha\bot)$ and $(s_0, \bot) \overset{\mu}{\underset{\mathcal{S}}{\to}} (p', \alpha'\bot)$. Since $\mathcal{S}$ is deterministic and has no ς-moves, Definition 5 gives $p = p'$ and $\alpha = \alpha'$. Thus, $(p, \alpha\bot) \overset{\ell}{\underset{\mathcal{S}}{\to}} (r, \gamma\bot)$. Since $(p, \ell, X, \textbf{fail})$ is a transition of $\mathcal{T}$, together with $(p, \alpha\bot) \overset{\ell}{\underset{\mathcal{S}}{\to}} (r, \gamma\bot)$, there are three cases. We show that all lead to a contradiction, thus showing that $\sigma \notin otr(\mathcal{S})$ as desired.

If $\ell \in L_c$ then (p, ℓ, W, r) is a transition of $\mathcal{S}$ for some $W \in \Delta_{\mathcal{S}}$. In this case we must have used Eq. (1) to insert $(p, \ell, X, \textbf{fail})$ in $T_{\mathcal{T}}$. But then we need $X = Z$ and $(p, \ell, Y, q) \notin T_{\mathcal{S}}$ for any $q \in S_{\mathcal{S}}$, $Y \in \Delta_{\mathcal{S}}$, and we get a contradiction. If $\ell \in L_i$ then $W = \sharp$, $(p, \ell, \sharp, r)$ is a transition of $\mathcal{S}$ and we must have used Eq. (3) to insert $(p, \ell, \sharp, \textbf{fail})$ in $T_{\mathcal{T}}$. But this requires $(p, \ell, \sharp, q) \notin T_{\mathcal{S}}$ for any $q \in S_{\mathcal{S}}$, and we reach a contradiction again. If $\ell \in L_r$ then (p, ℓ, W, r) is a transition of $\mathcal{S}$ where W is the first symbol in $\alpha\bot$. According to used Eq. (2) we now need $(p, \ell, W, q) \notin T_{\mathcal{S}}$ for any $q \in S_{\mathcal{S}}$.

Let $\mathcal{I} = \langle S_{\mathcal{I}}, I_{in}, L_I, L_U, \Delta_{\mathcal{I}}, T_{\mathcal{I}} \rangle$ be an arbitrary IUT. As argued above, we need $\mathcal{I}$ does not pass $\mathcal{T}$ if and only if $otr(\mathcal{I}) \cap T \neq \emptyset$, where $T = \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$. According to Definition 12, $\mathcal{I}$ does not pass $\mathcal{T}$ if and only if for some $\sigma \in (L_I \cup L_U)^\star$ we get $((t_0, q_0), \bot) \overset{\sigma}{\underset{\mathcal{T}\times\mathcal{I}}{\Rightarrow}} ((\textbf{fail}, q), \alpha\bot)$ where $((t_0, q_0), \bot)$ is an initial configuration of $\mathcal{T} \times \mathcal{I}$, $q \in S_{\mathcal{I}}$ and $\alpha = (X_1, Y_1) \cdots (X_n, Y_n) \in (\Delta_{\mathcal{T}} \times \Delta_{\mathcal{I}})^\star$, $n \geq 0$. Since **fail** is a sink state and transitions into **fail** are over symbols in L_U, we can say that $\mathcal{I}$ does not pass $\mathcal{T}$ if and only if

$$((t_0, q_0), \bot) \overset{\mu}{\underset{\mathcal{T}\times\mathcal{I}}{\Rightarrow}} ((p, r), \beta\bot) \overset{\ell}{\underset{\mathcal{T}\times\mathcal{I}}{\Rightarrow}} ((\textbf{fail}, q), \alpha\bot),$$

for some $\mu \in L^\star$, $\ell \in L_U$, $p \in S_{\mathcal{T}}$, $p \neq \textbf{fail}$, $r \in S_{\mathcal{I}}$, $\beta = (W_1, Z_1) \cdots (W_m, Z_m) \in (\Delta_{\mathcal{T}} \times \Delta_{\mathcal{I}})^\star$, $m \geq 0$.

First, we assume $otr(\mathcal{I}) \cap T \neq \emptyset$ and argue that $\mathcal{I}$ does not pass $\mathcal{T}$. We have $\sigma \in otr(\mathcal{I}) \cap T$ for some $\sigma \in (L_I \cup L_U)^\star$. Hence, $(q_0, \bot) \overset{\eta}{\underset{\mathcal{I}}{\to}} (r, \alpha\bot)$ with $\sigma = h_\varsigma(\eta)$ and $q_0 \in Q_{in}$. Since $\sigma \in \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$, Claim 2 gives $(t_0, \bot) \overset{\mu}{\underset{\mathcal{T}}{\to}} (\textbf{fail}, \beta\bot)$ with $\sigma = h_\varsigma(\mu)$ and $t_0 \in T_{in}$. We can now use Proposition 2 and write $((t_0, q_0), \bot) \overset{\sigma}{\underset{\mathcal{T}\times\mathcal{I}}{\Rightarrow}} ((\textbf{fail}, r), \gamma\bot)$. Since (t_0, q_0) is initial in $\mathcal{T} \times \mathcal{I}$, we see that $\mathcal{I}$ does not pass $\mathcal{T}$.

Lastly, assume that $\mathcal{I}$ does not pass $\mathcal{T}$. We get $\sigma \in (L_I \cup L_U)^\star$ and $((t_0, q_0), \bot) \underset{\mathcal{T}\times\mathcal{I}}{\overset{\sigma}{\Rightarrow}} ((\textsf{fail}, r), \gamma\bot)$ with (t_0, q_0) initial in $\mathcal{T} \times \mathcal{I}$. From Claim 3 we get $\sigma \in \overline{otr}(\mathcal{S}) \cap \big[otr(\mathcal{S})L_U\big]$. It is also clear that $((t_0, q_0), \bot) \underset{\mathcal{T}\times\mathcal{I}}{\overset{\eta}{\rightarrow}} ((\textsf{fail}, r), \gamma\bot)$ with $h_\varsigma(\eta) = \sigma$. From Proposition 3 we see that $(q_0, \bot) \underset{\mathcal{I}}{\overset{\mu}{\rightarrow}} (r, \gamma\bot)$, where $\gamma \in \Delta_{\mathcal{I}}^\star$ and $h_\varsigma(\mu) = \varsigma(\eta) = \sigma$. Thus, $(q_0, \bot) \underset{\mathcal{I}}{\overset{\sigma}{\Rightarrow}} (r, \gamma\bot)$ and, since $q_0 \in I_{in}$, we get $\sigma \in otr(\mathcal{I})$. Hence $\sigma \in otr(\mathcal{I}) \cap T$.

Now we have that $\mathcal{I}$ does not pass $\mathcal{T}$ if and only if $otr(\mathcal{I}) \cap T \neq \emptyset$, as needed.

5 Testing IOVPTS Models for ioco-like conformance

Given an IOVPFM which is complete for a given specification, we can test whether IUTs **ioco-like** conform to that specification. But first we define the notion of *balanced run*.

Let $\mathcal{V} \in \mathcal{VP}(L)$, and let p, q be states of $\mathcal{V}$. We say that a string $\sigma \in L^\star$ induces a *balanced run* from p to q in $\mathcal{V}$ if we have $(p, \bot) \underset{\mathcal{V}}{\overset{\sigma}{\rightarrow}} (q, \bot)$. The next theorem gives a decision procedure for testing **ioco-like** conformance.

Theorem 6 *Let $\mathcal{S} = \langle S_\mathcal{S}, \{s_0\}, L_I, L_U, \Delta_\mathcal{S}, T_\mathcal{S}\rangle \in \mathcal{IOVP}(L_I, L_U)$ be a deterministic specification, and let $\mathcal{I} = \langle S_\mathcal{I}, I_{in}, L_I, L_U, \Delta_\mathcal{I}, T_\mathcal{I}\rangle \in \mathcal{IOVP}(L_I, L_U)$ be an IUT. Then we can effectively decide whether $\mathcal{I}$* **ioco-like** *S holds. Further, if $\mathcal{I}$* **ioco-like** *S does not hold, we can find $\sigma \in otr(\mathcal{S})$, $\ell \in L_U$ that verify this condition,* i.e., *$\ell \in$* **out**$((q_0, \bot)$ **after** $\sigma)$ *for some $q_0 \in I_{in}$, and $\ell \notin$* **out**$((s_0, \bot)$ **after** $\sigma)$.

Proof To simplify the notation, let $L = L_I \cup L_U$. The proof of Lemma 2 indicates how to obtain a deterministic fault model $\mathcal{T} = \langle S_\mathcal{T}, T_{in}, L_U, L_I, \Delta_\mathcal{T}, T_\mathcal{T}\rangle$ such that $\mathcal{I}$ **ioco-like** $\mathcal{S}$ does not hold if and only if $((t_0, q_0), \bot) \underset{\mathcal{P}}{\overset{\sigma}{\Rightarrow}} ((\textsf{fail}, q), \alpha\bot)$ for some $\sigma \in L^\star$, where $\mathcal{P} = \mathcal{T} \times \mathcal{I}$ is the product IOVPTS, and (t_0, q_0) is an initial state of $\mathcal{P}$. So, in order to check for **ioco-like** conformance it suffices to check whether a configuration $((\textsf{fail}, q), \alpha\bot)$ is reachable from some initial configuration of $\mathcal{P}$. First, we modify $\mathcal{P}$ in a simple way in order to make this reachability problem more amenable.

Emptying the stack after reaching a state $(\textsf{fail}, q)$. For all states $(\textsf{fail}, q)$, add the internal transition $((\textsf{fail}, q), \varsigma, \sharp, f_1)$ to $\mathcal{P}$, where f_1 is a new state. Then, for all stack symbols W add the self-loops (f_1, b_1, W, f_1) to $\mathcal{P}$, where b_1 is a new pop symbol added to L. Next, add the transition $(f_1, b_1, \bot, f_2)$ to $\mathcal{P}$, where f_2 is another new state. Let $\mathcal{P}_1$ be the resulting IOVPTS obtained after these modifications to $\mathcal{P}$. Since $(\textsf{fail}, q)$ is a sink state in $\mathcal{P}$, it is easy to see that

(1) If $((t_0, q_0), \bot) \overset{\sigma}{\rightarrow} ((\textsf{fail}, q), \alpha\bot)$ in $\mathcal{P}$ then $((t_0, q_0), \bot) \overset{\mu}{\rightarrow} (f_2, \bot)$ in $\mathcal{P}_1$, where $\mu = \sigma\varsigma b_1^k$ with $k = |\alpha| + 1$.

(2) If $((t_0, q_0), \bot) \overset{\mu}{\rightarrow} (f_2, \bot)$ in $\mathcal{P}_1$ then $\mu = \sigma\varsigma b_1^k$ and $((t_0, q_0), \bot) \overset{\sigma}{\rightarrow} ((\textsf{fail}, q), \alpha\bot)$ in $\mathcal{P}$ for some $\alpha \in L^\star$ with $|\alpha| = k + 1$.

Assume that $\mathcal{P}$ has been transformed as described, so that we can always empty the stack after reaching a $(\textsf{fail}, q)$ state, for all states q of $\mathcal{I}$.

Eliminating pop moves on an empty stack. Let s_0 be yet a new state, a_2 a new push symbol, and Z_2 a new stack symbol. Add the self-loop (s_0, a_2, Z_2, s_0) to $\mathcal{P}$. Next we connect s_0 to all original initial states of $\mathcal{P}$ with internal transitions $(s_0, \varsigma, \sharp, s)$ where s is initial $\mathcal{P}$, and make s_0 the new, unique, initial symbol of $\mathcal{P}$. Finally, we replace any pop transition on an empty stack $(p, c, \bot, q)$ by the pop transition (p, c, Z_2, q). Let $\mathcal{P}_2$ be the new IOVPTS after these modifications to $\mathcal{P}$. Now we have

(1) If $(s, \bot) \overset{\sigma}{\rightarrow} (q, \bot)$ in $\mathcal{P}$ where s is one of its initial state, and if we have $0 \leq k \leq |\sigma|$ pop moves on the empty stack on this run, then by an induction on k we can show that $(s_0, \bot) \overset{a_2^k}{\rightarrow} (s_0, Z_2^k\bot) \overset{\varsigma}{\rightarrow} (s, Z_2^k\bot) \overset{\sigma}{\rightarrow} (q, \bot)$ in $\mathcal{P}_2$.

(2) If $(s_0, \bot) \overset{\sigma}{\rightarrow} (q, \bot)$ in $\mathcal{P}_2$, then an easy argument shows that $\sigma = a_2^k\varsigma\mu$, with $k \geq 0$, and we have $(s, \bot) \overset{\mu}{\rightarrow} (q, \bot)$ in $\mathcal{P}$, where s is the initial state in $\mathcal{P}$ and this run makes k pop moves on an empty stack.

Assume that the original product $\mathcal{P} = \mathcal{T} \times \mathcal{I}$ has been transformed into the IOVPTS $\mathcal{P}'$ after the modifications that allow us to empty the stack after reaching a $(\textsf{fail}, q)$ state and to avoid pop moves on an empty stack. Then, we have $((t_0, q_0), \bot) \underset{\mathcal{P}}{\overset{\sigma}{\rightarrow}} ((\textsf{fail}, q), \alpha\bot)$, with (t_0, q_0) as the initial state in $\mathcal{P}$, if and only if we have $(s_0,\perp) \xrightarrow[\mathcal{P}']{\mu} (f_2,\perp)$, where $\mu = a_2^k \varsigma\sigma\varsigma b_1^n$ for some $k \geq 0$ and $n \geq 1$. Now, from the definition, we have $((t_0,q_0),\perp) \overset{\eta}{\underset{\mathcal{P}}{\Rightarrow}} ((\textsf{fail},q),\alpha\perp)$ if and only if $((t_0,q_0),\perp) \xrightarrow[\mathcal{P}]{\sigma} ((\textsf{fail},q),\alpha\perp)$ where $\eta = h_\varsigma(\sigma)$. Thus, $((t_0,q_0),\perp) \overset{\eta}{\underset{\mathcal{P}}{\Rightarrow}} ((\textsf{fail},q),\alpha\perp)$ if and only if we have a balanced run μ from s_0 to f_2 in $\mathcal{T}$ and $\eta = h_{\{a_2,b_1,\varsigma\}}(\mu)$, *i.e.*, we get η from μ by erasing all occurrences of a_2, b_1 and ς. Putting it together, we have: $\mathcal{I}$ **ioco-like** $\mathcal{S}$ does not hold if and only if $(s_0,\perp) \xrightarrow[\mathcal{T}]{\mu} (f_2,\perp)$, and $h_{\{a_2,b_1,\varsigma\}}(\mu)$ is a string that corroborates this fact.

We have reduced the **ioco-like** conformance test to the following problem: given two states p and q of an IOVPTS, find a string σ that induces a balanced run from p to q, or indicate that such a string does not exist. Next, we solve this problem.

The following construction was inspired from ideas of previous works [14, 19]. Now, it is sufficient to consider the underlying VPTS. So, let $\mathcal{P} = \langle Q, Q_{in}, L, \Gamma, \rho\rangle$ be a VPTS given by an incidence vector P of transitions, indexed by Q, where $P[p]$ points to a list of all transitions $(p,x,Z,q) \in \rho$ where p is the source state. We assume that $\mathcal{P}$ has no pop transitions on the empty stack, that is, of the form $(p,x,\perp,q)$ where x is a pop symbol. Algorithm 1 shows the pseudo-code.

Algorithm 1: Checking for balanced runs in a VPTS $\mathcal{P} = \langle Q, Q_{in}, L, \Gamma, \rho\rangle$

```
    Data: Given: a vector $P$, where $P[p]$ is a list of all $(p,a,Z,q) \in \rho$; states $s_i$, $s_e$.
    Data: Assumptions: $\mathcal{P}$ has no transition on the empty stack and $s_i \neq s_e$.
    Data: Uses: vectors $In$, $Out$ indexed by $Q$; matrix $R$ indexed by $Q \times Q$; queue $V$.
    Result: Check for a BR from $s_i$ to $s_e$; if there is one find a string that induces it.
    // Initialize $V$, $R$, $In$ and $Out$
 2  $V = null$
 3  forall $p, q$ in $Q$ do {$R[p,q] = 0$}
 4  forall $p$ in $Q$ do { $In[p] = null$; $Out[p] = null$ }
 5  forall $p$ in $Q$ do
 6      forall $(p,a,Z,q)$ in $P[p]$ do
 7          if $a \in L_i \cup \{\varsigma\}$ and $p \neq q$ and $R[p,q] = 0$ then
                {$R[p,q] = [p,a,q]$; add $(p,q)$ to $V$}          // simp & inter transitions
 8          else if $a \in L_r$ then add $(a,Z,q)$ to $Out[p]$         // pop transition
 9          else add $(p,a,Z)$ to $In[q]$                          // push transition
10              forall $(q,b,W,r) \in P[q]$ do              // BR from push & pop transitions
11                  if $W = Z$ and $b \in L_r$ and $p \neq r$ and $R[p,r] = 0$ then
                        {$R[p,r] = [a,q,q,b]$; add $(p,r)$ to $V$ }
    //
    // Main loop
 1
12  while $V \neq null$ and $R[s_i,s_e] = 0$ do
13      Remove $(p,q)$ from $V$                                    // We have a BR from $p$ to $q$
14      forall $s$ in $Q$ do                                        // new BR from s to q
15          if $R[s,p] \neq 0$ and $s \neq q$ and $R[s,q] = 0$ then { $R[s,q] = [s,p,q]$; add $(s,q)$ to $V$ }
16      forall $t$ in $Q$ do                                        // new BR from p to t
17          if $R[q,t] \neq 0$ and $p \neq t$ and $R[p,t] = 0$ then { $R[p,t] = [p,q,t]$; add $(p,t)$ to $V$ }
18      forall $(s,a,Z)$ in $In[p]$ do
19          forall $(b,W,t)$ in $Out[q]$ do            // push $Z$ from $s$ to $p$, pop $Z$ from $q$ to $t$
20              if $W = Z$ and $s \neq t$ and $R[s,t] = 0$ then { $R[s,t] = [a,p,q,b]$; add $(s,t)$ to $V$ }
    //
    // Issue the verdict
21  if $R[s_i,s_e] = 0$ then Print THERE ARE NO BALANCED RUNS FROM $s_i$ TO $s_e$
22  else Print A STRING THAT INDUCES A BALANCED RUN FROM $s_i$ TO $s_e$ (BETWEEN | |): | getstring $(s_i,s_e)$ |
    //
23  Function getstring$(p,q)$:                                      // Print the string
24      switch $R[p,q]$ do
25          case $[p,a,q]$ do Print "$a$"
26          case $[p,s,q]$ do { getstring $(p,s)$; getstring $(s,q)$ }
27          case $[a,p,q,b]$ do if $p \neq q$ then { Print "$a$"; getstring $(p,q)$; Print "$b$" } else { Print "$a$";
              Print "$b$" }
28
```

We will use two vectors of pointers, In and Out, both indexed by Q, and a queue V. The entry $In[p]$ will point to a list of triples (s,a,Z) corresponding to transitions (s,a,Z,p) where $a \in L_c$, that is, p as the target state of a push transition. Likewise, an entry in $Out[p]$ will point to a list of triples (a,Z,s) corresponding to transitions (p,a,Z,s) where $a \in L_r$, that is, p is the source state of a pop transition. We will also need a square matrix R, indexed by $Q \times Q$, where $R[p,q]$ will contain: (i) $[a,p,q,b]$, or (ii) $[p,c,q]$, or (iii) $[p,s,q]$, or (iv) 0, where $a \in L_c$, $b \in L_r$, $c \in L_i \cup \{\varsigma\}$, and p, q, s are states. The general idea is that, when $R[p,q] \neq 0$ then it will code for a string σ that induces a balanced run from p to q.

We now examine Algorithm 1. Lines 1–10 initialize V, In, Out and R. At line 6, note that a transition $(p,a,\sharp,q)$ immediately induces a balanced run from p to q. At line 10, we collect a simple balanced run from

p to r that is induced by a push transition (p, a, Z, q) and a pop transition (q, b, Z, r). In the main loop, lines 11–19, removing (p, q) from V indicates that we already have a string, say σ, that induces a balanced run from p to q. At lines 13–14, a string μ that induces balanced run from s to p is encoded in $R[s, p]$. Hence, $\mu\sigma$ induces a balanced run from s to q. If we still do not have a balanced run from s to q, we can now encode the string $\mu\sigma$ in $R[s, q]$ and move the pair(s, q) to V so that it can be examined later. Lines 15–16 do the same, but now we encode in $R[p, t]$ a string that induces a balanced run from p to t. At lines 17–19, we search for a push transition (s, a, Z, p) and a matching pop transition (q, b, Z, t) and, when successful, we encode $a\sigma b$ as a string that induces a balanced run from s to t. The cycle repeats until saturation when $V = null$, or until we find a balanced run from s_i to s_e, as requested. Lines 22–26 list a recursive procedure that extracts the string encoded in $R[p, q] \neq 0$.

5.1 Correctness and Complexity

First we argue for correctness.

Theorem 7 *Let $\mathcal{P} = \langle Q, Q_{in}, L, \Gamma, \rho\rangle$ be a VPTS with no transitions of the form $(p, a, \perp, q)$ in ρ. Also let s_i, $s_e \in Q$, with $s_i \neq s_e$. Suppose $\mathcal{P}$, s_i, s_e are input to Algorithm 1. Then it stops and returns a string $\sigma \in L^\star$ such that $(s_i, \perp) \underset{\mathcal{P}}{\overset{\sigma}{\to}} (s_e, \perp)$, or it indicates that such a string does not exist.*

Proof A VPTS $\mathcal{P} = \langle Q, Q_{in}, L, \Gamma, \rho\rangle$ with no transitions of the form $(p, a, \perp, q)$ in ρ, and two states s_i, s_e with $s_i \neq s_e$, are input to Algorithm 1.

At lines 1 and 2 we start with $V = null$ and $R[p, q] = 0$ for all pairs (p, q). Inspecting lines 6, 10, 14, 16 and 19, we see that a pair (p, q) is added to V only when we currently have $R[p, q] = 0$ and, when(p, q) enters V we immediately set $R[p, q]$ to some nonzero value. Further, at no other point in the main loop, at lines 11–19, we reset $R[p, q]$ to zero. Hence, a pair (p, q) can enter V at most once and, therefore, the main loop at line 11 must terminate. So, Algorithm 1 always stops.

Next we claim that, at any point during the execution of the algorithm, if $R[p, q] \neq 0$ then it codes for a string that induces a balanced run from p to q. This is immediate from the initialization lines 6 and 10. Proceeding inductively, assume that this property holds after a number of executions of the main loop. At line 14, we have removed (p, q) from V, and so we now have $R[p, q] \neq 0$ because (p, q) entered V in a previous iteration. At that moment we made $R[p, q] \neq 0$ and then, inductively, it codes for a string σ such that $(p, \perp) \overset{\sigma}{\to} (q, \perp)$. Now, at line 14 we require $R[s, p] \neq 0$ so that, inductively, it also codes for a string μ such that $(s, \perp) \overset{\mu}{\to} (p, \perp)$. Composing, we get $(s, \perp) \overset{\mu\sigma}{\to} (q, \perp)$, and so by making $R[s, q] \neq 0$ code for the string $\mu\sigma$, we extend the induction in this case. The reasoning at line 16, is very similar. We now look at line 19. At that point we have a push transition (s, a, Z, p), a pop transition (q, b, Z, t), and $(p, \perp) \overset{\sigma}{\to} (q, \perp)$ for some $\sigma \in (L \cup \{\varsigma\})^\star$. Recall that the algorithm assumes that the given VPTS $\mathcal{P}$ has no pop transitions on the empty stack. With this hypothesis, we claim the following general property of $\mathcal{P}$:

> If $(p, \alpha_1\perp) \underset{\mathcal{P}}{\overset{\sigma}{\to}} (q, \alpha_2\perp)$ with p, $q \in Q$ and $\alpha_1, \alpha_2 \in \Gamma^\star$, then for all $\beta_1, \beta_2 \in \Gamma^\star$ we also have $(p, \alpha_1\beta_1\perp) \underset{\mathcal{P}}{\overset{\sigma}{\to}} (q, \alpha_2\beta_2\perp)$.
>
> A simple proof can be obtained by induction on $|\sigma| \geq 0$.

With $\alpha_1 = \alpha_2 = \varepsilon$, $\beta_1 = \beta_2 = Z$, from $(p, \perp) \overset{\sigma}{\to} (q, \perp)$ we get $(p, Z\perp) \overset{\sigma}{\to} (q, Z\perp)$. Now we have $(s, \perp) \overset{a}{\to} (p, Z\perp) \overset{\sigma}{\to} (q, Z\perp) \overset{b}{\to} (t, \perp)$, so that making $R[s, t]$ code for the string $a\sigma b$ also extends the induction after line 19 is passed. Since we have completed one more iteration of the main loop, we see that upon termination of the main loop, if we have $R[s_i, s_e] \neq 0$, then we do have a string that induces a balanced run from s_i to s_e. Moreover, it easy to see that the simple recursive call `getstring(`s_i, s_e`)` at line 21 does correctly extract one such string.

Next, we argue in the other direction. Suppose that the main loop terminates with $V = null$. Then we claim that for all pairs (p, q), with $p \neq q$, if $R[p, q] = 0$ then there is no string capable of inducing a balanced run from p to q. For the sake of contradiction, assume that the main loop terminates with $V = null$, and we have $p \neq q$, $R[p, q] = 0$, and a string σ such that $(p, \perp,) \overset{\sigma}{\to} (q, \perp)$. Among all such pairs, choose one for which $|\sigma|$ is minimum. Since $p \neq q$, we need $|\sigma| \geq 1$. If $|\sigma| = 1$, then we need a transition $(p, \sigma, \sharp, q)$ in ρ. But then, at line 6, we make $R[p, q] = [p, \sigma, q]$ and it is never rest to 0 again. This is a contradiction, and we can assume $|\sigma| \geq 2$.

Now there are two cases, depending on the configurations that occur between $(p, \perp)$ and $(q, \perp)$ in the run $(p, \perp) \overset{\sigma}{\to} (q, \perp)$:

Case 1: A configuration $(r, \perp)$ occurs in the run $(p, \perp) \xrightarrow{\sigma} (q, \perp)$. Write $(p, \perp) \xrightarrow{\sigma_1} (r, \perp) \xrightarrow{\sigma_2} (q, \perp)$, with $\sigma = \sigma_1\sigma_2$. We know that $\sigma_1 \neq \varepsilon \neq \sigma_2$ because $\mathcal{P}$ has no transitions on the empty stack. If $p = r$ we get $(p, \perp) \xrightarrow{\sigma_2} (q, \perp)$. Since $|\sigma_2| < |\sigma|$, the minimality of $|\sigma|$ forces $R[p, q] \neq 0$, a contradiction. Similarly, $q = r$ also leads to a contradiction.

Now, assume $p \neq r \neq q$. Since $|\sigma_1| < |\sigma|$ and $|\sigma_2| < |\sigma|$, when the main loop terminates with $V = \mathit{null}$ we must have $R[p, r] \neq 0$ and $R[r, q] \neq 0$. Moreover, for this to happen, both (p, r) and (r, q) were added to V. Suppose that (p, r) is removed from V before (r, q). Then, at the iteration of the main loop when (r, q) is removed from V we have $R[p, r] \neq 0$ and $p \neq q$. Hence, at line 14, since $R[p, q] = 0$, we make $R[p, q] = [p, r, q]$ and, since it is never reset to 0 again, we have a contradiction. If (r, q) is removed first from V, the reasoning is the same using line 16. So, this case can not happen.

Case 2: A configuration $(r, \perp)$ does not occur in the run $(p, \perp) \xrightarrow{\sigma} (q, \perp)$. Then, we must have a push transition (p, x, Z, s) with $\sigma = x\sigma_1$, and we are left with $(p, \perp) \xrightarrow{x} (s, Z\perp) \xrightarrow{\sigma_1} (q, \perp)$. If $|\sigma_1| = 1$, we need a pop transition (s, y, Z, q) and now line 10 makes $R[p, q] = [x, q, q, y]$, a contradiction. Hence $|\sigma_1| \geq 2$. Since no configuration of the form $(u, \perp)$ occurs in the run over σ, we must have a pop transition (t, y, Z, q) and $(p, \perp) \xrightarrow{x} (s, Z\perp) \xrightarrow{\mu} (t, Z\perp) \xrightarrow{y} (q, \perp)$, with $\sigma = x\mu y$.

Next we claim that in any VPTS $\mathcal{S} = \langle S, S_{in}, L, \Gamma, T\rangle$ if a run does not shorten the initial stack, then that stack can be replaced by any other. More precisely,

Let $p, q \in S$, $\sigma \in (L \cup \{\varsigma\})^\star$ and $\alpha \in \Gamma^\star$ with $(p, \alpha\perp) \xrightarrow{\sigma} (q, \alpha\perp)$. Assume that a configuration (u, γ), with $|\gamma| < |\alpha|$, does not occur in that run over σ. Then, for any $\beta \in \Gamma^\star$ we also have $(p, \beta\perp) \xrightarrow{\sigma} (q, \beta\perp)$.

An easy induction over $|\sigma| \geq 0$ gives the result.

Recall that we already have $(s, Z\perp) \xrightarrow{\mu} (t, Z\perp)$ and a configuration $(u, \perp)$ dos not occur on the run over μ since it can not occur on a run over σ. Using the claim we get $(s, \perp) \xrightarrow{\mu} (t, \perp)$. Now, since $|\mu| < |\sigma|$, the minimality of $|\sigma|$ says that when the main loop terminates with $V = \mathit{null}$, we must have $R[s, t] \neq 0$. But then, at some moment (s, t) was added to V. Since the main loop terminates with $V = \mathit{null}$, at some iteration we have removed (s, t) from V. Note that we have a push transition (p, x, Z, s) and a pop transition (t, y, Z, q). Hence, line 19 says that we will set $R[p, q] = [x, s, t, y]$ and, since $R[p, q]$ is never reset, we see that the main loop terminates with $R[p, q] \neq 0$, which is a contradiction.

Since both cases lead to contradictions, we get that when the main loop terminates with $V = \mathit{null}$ and $R[s_i, s_e] = 0$, then there is no string capable of inducing a balanced run from s_i to s_e. Hence, line 20 correctly reports the inexistence of any such strings.

Thus, lines 20–21 always report as expected, and Algorithm 1 is correct.

Now we examine the complexity of our testing approach.

Theorem 8 *Let $\mathcal{S} \in \mathcal{IOVP}(L_I, L_U)$ be a deterministic specification with n_s states and m_s transitions, and let $\mathcal{I} \in \mathcal{IOVP}(L_I, L_U)$ be an IUT with n_i states and m_i transitions. Then there is a procedure, with worst case asymptotic polynomial time complexity bounded by $\mathcal{O}(n_s^3 n_i^3 + n_s^2 m_s^4 m_i^2)$, that verifies whether $\mathcal{I}$* **ioco-like** *$\mathcal{S}$. Moreover, if $\mathcal{I}$* **ioco-like** *S does not hold, the procedure finds an input string that proves this condition.*

Proof Write $L = L_I \cup L_U$, $|L| = \ell$ and let g_s, g_i be the number of stack symbols in $\mathcal{S}$ and $\mathcal{I}$, respectively. We now follow the argument in the proof of Theorem 6.

First, the fault model $\mathcal{T}$ is constructed in Lemma 2. From Equations (1)–(3) we see that $n_t = n_s + 1$ and $m_t \leq m_s + n_s g_s \ell$, where n_t and m_t are the number of states and transitions in $\mathcal{T}$, respectively. It is easy to see that $\mathcal{T}$ can be effectively constructed from $\mathcal{S}$ by an algorithm with worst case time complexity bounded by $\mathcal{O}(m_s + n_s g_s \ell)$. We can safely assume $m_s \geq g_s$ and $m_s \geq \ell$. Hence, m_t and n_t can be bounded by $\mathcal{O}(n_s m_s^2)$ and $\mathcal{O}(n_s)$, respectively, and the worst case time complexity to obtain $\mathcal{T}$ can also be bounded by $\mathcal{O}(n_s m_s^2)$.

Next, we construct the product $\mathcal{P} = \mathcal{T} \times \mathcal{I}$. Let n_p, m_p and g_p be the number of states, transitions and stack symbols in $\mathcal{P}$, respectively. Using Definition 6, and since $\mathcal{T}$ has no ς-moves, we see that $n_p = n_t n_i$, $m_p \leq m_t m_i + n_t m_i$, and $g_p = g_s g_i$. As before, a simple algorithm with worst case time complexity bounded by $\mathcal{O}(m_t m_i + n_t m_i)$ can construct $\mathcal{P}$ given $\mathcal{T}$ and $\mathcal{I}$. Hence, n_p, m_p and g_p can be bounded by $\mathcal{O}(n_s n_i)$, $\mathcal{O}(n_s m_s^2 m_i)$ and $\mathcal{O}(m_s m_i)$, respectively, and the worst case time complexity to obtain $\mathcal{P}$ can be bounded by $\mathcal{O}(n_s m_s^2 m_i)$.

Finally, Theorem 6 requires that we modify $\mathcal{P}$ to an IOVPTS with an underlying VPTS $\mathcal{A} = \langle S_a, \{s_0\}, L_a, \Gamma_a, \rho_a \rangle$ with the property that $\mathcal{I}$ **ioco-like** S does not hold if and only if we have $(s_0, \perp) \xrightarrow[\mathcal{A}]{\mu} (f, \perp)$ for some $\mu \in (L_a \cup \{\varsigma\})^\star$, where f is a specific state in S_a with $s_0 \neq f$. Moreover, if such is the case, the final steps in the proof of Theorem 6 indicate how to obtain the desired string σ that proves that $\mathcal{I}$ **ioco-like** S fails. Let $n_a = |S_a|$ and $m_a = |\rho_a|$. From the proof of Theorem 6, it is easy to get $n_a = n_p + 3$ and $m_a = m_p + 2n_i + g_p + 2$. Also, a simple procedure, running in worst case time complexity $\mathcal{O}(m_a + n_a)$, can be used construct the VPTS $\mathcal{A}$ given the product $\mathcal{P}$. Then, n_a can be bounded by $\mathcal{O}(n_s n_i)$, m_a can be bounded by $\mathcal{O}(n_s m_s^2 m_i)$, and the worst case time complexity to construct $\mathcal{A}$ can be bounded by $\mathcal{O}(n_s m_s^2 m_i)$.

The final step is to submit $\mathcal{A}$ and the two states s_0, f to Algorithm 1. Theorem 7 guarantees that Algorithm 1 correctly produces a desired string or indicates that no such string exists.

We now derive an asymptotic upper bound on the number of steps required for Algorithm 1 in the worst case. Clearly, the number of steps for lines 1–3 can be bounded by $\mathcal{O}(n_a^2)$.

For each state $p \in S_a$, let s_p, t_q be the number of transitions in $\mathcal{A}$ that have p as a source and q as a target state, respectively. Thus, $\sum_{p \in Q} s_p \leq m_a$ and $\sum_{p \in Q} t_p \leq m_a$ for all $p \in S_a$. The total number of steps pertaining to lines 4–10 can be bound by

$$\sum_{p \in S_a} \big(s_p \sum_{q \in S_a} t_q\big) \leq \sum_{p \in S_a} (s_p m_a) = m_a \sum_{p \in S_a} s_p \leq m_a^2.$$

From the proof of Theorem 7, we have that each pair (p, q) can enter the queue V at most once. Hence, a state p will appear in a pair (p, q) removed from V at most n_a times. Since the number of steps at each execution of lines 13–14 can be bounded by $\mathcal{O}(n_a^2)$, the total effort spent for lines 13–14 is bound by $\mathcal{O}(n_a^3)$. Likewise for lines 15–16. Now we bound the total number of steps for lines 17–19. For each pair (p, q) removed from V, the cost relative to lines 17–19 is $\mathcal{O}(t_p s_q)$. Since each pair of (p, q) can enter V at most once, the total cost is bound by

$$\sum_{p,q \in Q} t_p s_q = \sum_{p \in Q} t_p \big(\sum_{q \in Q} s_q\big) \leq \sum_{p \in Q} (t_p m_a)$$

$$\text{and} \quad \sum_{p \in Q} (t_p m_a) = m_a \sum_{p \in Q} t_p \leq m_a^2.$$

Hence, the total number of steps to execute the main loop at lines 12–19 is bound by $\mathcal{O}(n_a^3 + m_a^2)$. We can now conclude that the total number of steps to execute Algorithm 1 is bound by $\mathcal{O}(n_a^3 + m_a^2)$. Using the previously computed values, we see that a worst case asymptotic time complexity for Algorithm 1 is bounded by $\mathcal{O}(n_s^3 n_i^3 + n_s^2 m_s^4 m_i^2)$. Since this bound dominates all the preprocessing steps needed to construct $\mathcal{T}$, $\mathcal{P}$ and $\mathcal{A}$, the overall worst case time complexity of the **ioco-like** checking procedure is $\mathcal{O}(n_s^3 n_i^3 + n_s^2 m_s^4 m_i^2)$.

In some practical situations we may assume that the number of stack and alphabet symbols as constants, for any specifications and IUTs models that will be considered. In these cases, the number of transitions of the fault model $\mathcal{T}$ can be bounded by $\mathcal{O}(m_s)$, in the proof of Theorem 8. As a consequence, the worst case time complexity for Algorithm 1 can be seen to be bounded by $\mathcal{O}(n_s^3 n_i^3 + m_s^2 m_i^2)$.

5.2 Example: a Drink Dispensing Machine

Now we want to apply the previous results in a more realistic setting where a drink dispensing machine is operated. Because the overall testing procedure is not yet fully implemented on software, the example has to be somewhat contrived, so that we can proceed manually. So we show only some possible interactions that may occur. When dealing with a practical implementation, of course, many other situations could be represented, making the model as complex as required by the testing requirements one is dealing with.

We first describe the drink dispensing machine and its specification IOVPTS model. In the sequel we construct the fault model for the given specification and then we test some possible IUTs for **ioco-like** conformance.

5.2.1 A Drink Dispensing Machine

In a typical drink dispensing machine, a customer puts in some money and then order the desire beverage. After choosing the beverage, the right amount of money will be charged and the machine will dispense the chosen drink. If the amount of money was in excess, the customer can ask for the balance. If the amount of money already in the machine is not enough, the customer has to add more money or the customer can decide to get a full refund. Usually, real machines accept several payment methods such as cash and credit cards. In order to ease our modeling we specify that only unit coins can be used for payment.

The complete IOVPTS specification model $\mathcal{S} = \langle S_\mathcal{S}, S_{in}, L_I, L_U, \Delta_\mathcal{S}, T_\mathcal{S}\rangle$ is depicted in Figure 5, where $L_I = \{coi, rch, crd, wtr, tea, cof, deb\}$ is the set of input events and $L_U = \{chg, dwt, dte, dco\}$ is the set of

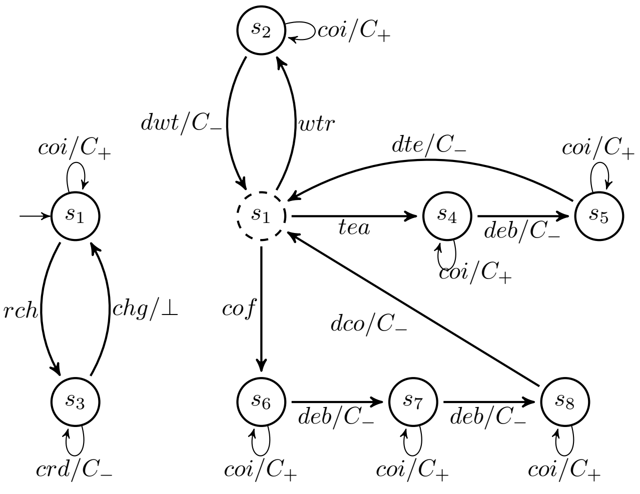


Figure 5: A drink dispensing machine $\mathcal{S}$.

output events. The alphabet $L = L_I \cup L_U$ is partitioned into the set of push events $L_c = \{coi\}$, the set of pop events $L_r = \{crd, chg, deb, dwt, dte, dco\}$ and the set of simple events $L_i = \{rch, wtr, tea, cof\}$. We have split state s_1 to make the figure clearer. Recall Remark 1 for the notation. The underlying VPTS is $\mathcal{A}_\mathcal{S} = \langle S_\mathcal{S}, S_{in}, L, \Delta_\mathcal{S}, T_\mathcal{S}\rangle$.

The system starts at state s_1 where the customer can either insert coins into the machine — event labeled *coi* —, request his change — event labeled *rch* —, or ask for a drink, namely, label *wtr* for water, label *tea* for tea, or label *cof* for coffee. Inserting coins is represented by the self-loop labeled coi/C_+ at state s_1. Note that the pushdown stack keeps track of the number of coins inserted into the machine. At state s_1 the customer can also request a refund, or the remaining change, by activating the *rch* event. The machine will then return the correct balance via the pop self-loop crd/C_- at state s_3.

The customer orders a drink by pushing the button for water, tea, or coffee, moving the machine to states s_2, s_4 and s_6, respectively. The price associated to water is one coin, for tea it is two coins and for coffee it is three coins. When the customer asks for water, the pop transition dwt/C_- is taken, returning to state s_1, and the correct charge is applied subtracting one coin from the total amount. The event *dwt* indicates that water has been dispensed. However, if not enough coins have been inserted, the transition from state s_2 back to state s_1 is blocked. The customer can proceed by inserting more coins using the self-loop coi/C_+ at state s_2. The behavior when ordering tea or coffee follows similar paths. For simplicity, once the customer has made a commit to order some of the beverages, the machine will wait until enough coins have been inserted to pay for the chosen drink.

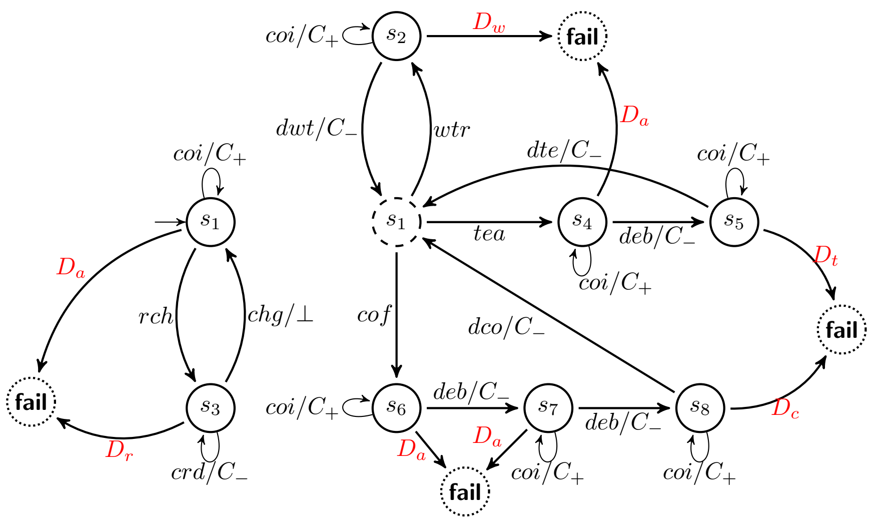
Figure 6 depicts the fault model $\mathcal{T}$ that is constructed for the specification $\mathcal{S}$ using Lemma 2. We have split the **fail** state in order keep the figure uncluttered. The sets D_ℓ, for $\ell \in \{a, r, w, t, c\}$, collect the label of several transitions, as indicated in the figure caption. Note that $L_U \cap L_c = \emptyset = L_U \cap L_i$, so that the only transitions into the **fail** state we need to check are those over the symbols of $L_U \cap L_r = \{chg, dwt, dte, dco\}$ together with stack symbols in $\{C, \perp\}$. For example, the set D_a denotes transitions to the **fail** state with pairs (x, y) for all $x \in L_U \cap L_r$ and all $y \in \{C, \perp\}$.

5.2.2 Testing Some Implementations

In this subsection, we examine some implementations, and test them for **ioco-like** conformance against the specification $\mathcal{S}$ depicted in Figure 5, and whose fault model $\mathcal{T}$ is shown in Figure 6.

Our first example is an IUT $\mathcal{I}_a$, depicted in Figure 7, where coffee is wrongly charged at two coins only. Here we notice that the state s_8 of $\mathcal{S}$ is missing, so that in the IUT $\mathcal{I}_a$ we have a self-loop $(s_7, deb/C_-, s_7)$ instead of the transition $(s_7, deb/C_-, s_8)$ as in the specification model. In this case a fault can occur and the machine may charge less for a cup of coffee.

Consider the sequence of events $\eta = \mu\, dco$, where $\mu = coi\, coi\, cof\, deb$. The customer has inserted only two coins and ordered coffee, and still the machine may deliver a cup of coffee. It is easy to see that η leads $\mathcal{T}$ to the **fail** state, while $\mathcal{I}_a$ reaches state s_1. Again, from Lemma 2 we obtain that $\mathcal{I}_a$ **ioco-like** $\mathcal{S}$ does not hold.



$$
\begin{aligned}
D_a &= \{dwt/C_-, dte/C_-, dco/C_-, chg/C_-, dwt/\perp, dte/\perp, dco/\perp, chg/\perp\}\\
D_r &= \{dwt/C_-, dte/C_-, dco/C_-, chg/C_-, dwt/\perp, dte/\perp, dco/\perp\}\\
D_w &= \{dte/C_-, dco/C_-, chg/C_-, dwt/\perp, dte/\perp, dco/\perp, chg/\perp\}\\
D_t &= \{dwt/C_-, dco/C_-, chg/C_-, dwt/\perp, dte/\perp, dco/\perp, chg/\perp\}\\
D_c &= \{dwt/C_-, dte/C_-, chg/C_-, dwt/\perp, dte/\perp, dco/\perp, chg/\perp\}
\end{aligned}
$$

Figure 6: The fault model $\mathcal{T}$ for $\mathcal{S}$.

In this example we have $dco \in L_U$ and $dco \in$ **out**$((s_1, \perp)$ **after** $\mu)$ in $\mathcal{I}_a$, while $dco \notin$ **out**$((s_1, \perp)$ **after** $\mu)$ in $\mathcal{S}$. So, from Definition 9, we can declare that $\mathcal{I}_a$ **ioco-like** $\mathcal{S}$ does not hold. In the product $\mathcal{T} \times \mathcal{I}_a$, we get the corresponding run

$$((s_1, s_1), \perp) \overset{\eta}{\Rightarrow} ((\textbf{fail}, s_1), \perp).$$

Notice that, in this same IUT $\mathcal{I}_a$, coffee could be charged more than three coins, *i.e.*, the machine may subtract more than three coins before dispensing a cup of coffee, when the user has inserted more than three coins before asking for the cup of coffee.

Now, we turn to IUT $\mathcal{I}_b$, obtained from $\mathcal{S}$ by adding the extra self-loop $(s_5, deb/C_-, s_5)$ to Figure 5. This allows the machine to subtract any number of extra coins after the customer has ordered a drink, given that more than enough coins have been inserted. Consider the sequence of events $coi\, coi\, coi\, tea\, deb\, deb\, dte\, rch\, chg$, signaling that the customer initially has inserted three coins, then decided to order tea. According to the IUT $\mathcal{I}_b$, however, when requesting the remaining change the customer gets no coins back, and the net effect was that the customer was charged three coins for a cup of tea. However, even in face of that mistake, we show below that $\mathcal{I}_b$ does conform to the specification $\mathcal{S}$.

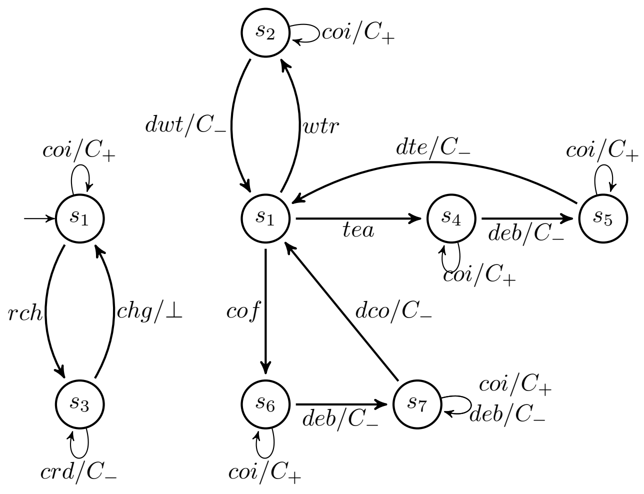


Figure 7: An IUT $\mathcal{I}_a$ charges wrong.

Recall the original specification $\mathcal{S}$. We note that $\mathcal{I}_b$ differs from $\mathcal{S}$ only by the extra transition at state s_5. Further, for each symbol $x \in L_U$ and state s_i, there is at most one transition out of s_i on x, both in $\mathcal{I}_b$ and in $\mathcal{S}$. Reasoning more formally, it is easy to see that for any sequence of events σ a simple induction on $|\sigma| \geq 0$ shows that if we have $(s_i, \alpha\bot) \in (s_1, \bot)$ **after** σ in $\mathcal{S}$ and $(s_j, \beta\bot) \in (s_1, \bot)$ **after** σ in $\mathcal{I}_b$, then $i = j$ and $\alpha = \beta$. According to Definition 9, if $\mathcal{I}_b$ **ioco-like** $\mathcal{S}$ did not hold, we would need $\ell \in L_U$ and a sequence σ such that $\ell \in$ **out**$((s_i, \alpha\bot))$ where $(s_i, \alpha\bot) \in (s_1, \bot)$ **after** σ in $\mathcal{I}_b$ and $\ell \notin$ **out**$((s_j, \beta\bot))$ where $(s_j, \beta\bot) \in (s_1, \bot)$ **after** σ in $\mathcal{S}$. Since the transitions of $\mathcal{S}$ and $\mathcal{I}_b$ are identical, except at state s_5, we conclude that $s_i = s_j = s_5$ and $\ell = deb$. Because $deb \notin L_U$ we reached a contradiction, and must conclude that $\mathcal{I}_b$ **ioco-like** $\mathcal{S}$ does hold.

We note that $deb \notin L_U$ was crucial to the preceding argument. In fact, if we move *deb* from L_I to L_U, then we clearly would get that $\mathcal{I}_b$ **ioco-like** $\mathcal{S}$ fails. This is because the nature of the **ioco-like** relation checks only that the IUT *may not* emit any *output symbol* that was not enabled in the specification, after they both experience any sequence of events that runs on the specification. On the other hand, the definition of the **ioco-like** relation says nothing about *input symbols* that may be emitted by the IUT and the specification after a common run in both models.

6 Concluding Remarks

Testing conformance of reactive systems implementations is usually a hard task, due to the intrinsic nature of these systems, which allows for the asynchronous interactions of messages with an external environment. In such situations, the use of rigorous approaches capable of generating test suites for these models is indicated.

Previous studies focused on simple systems which have access only to a finite memory, represented by its states, *e.g.*, LTS models. Here we studied a more powerful class of reactive systems, those that can make use of a potentially infinite memory, in the form of a pushdown stack. We extended the classical notion of **ioco**-conformance to cope with this new formalism that can make use of a potentially infinite memory. Essentially, this conformance relation still says that the implementation can only emit an output signal that is already present in the specification, after a common exchange of symbols has taken place in both models, but now both having access to a pushdown stack.

We developed, and proved correct, polynomial time algorithms that can be used to generate complete test suites and that can verify whether any implementation **ioco**-conforms to a given specification. In common practical situations the algorithms exhibit asymptotic worst case time complexity that can be bounded by $\mathcal{O}(n^3 + m^2)$ where n and m are proportional to the product of the number of states and transitions, respectively, present in the implementation and in the specification.

As discussed in Section 2 other works have also investigated recursive systems where a stack memory is present. They are, in general, based on classes of formal models which are proper subsets of Visibly Pushdown Languages, such as the class of Dyck languages. Therefore our approach is more complete and general in the sense that those systems and their respective models treated by our framework are also more powerful.

As additional work one could implement into a prototype the theoretical ideas developed here, and test the prototype with models that represent more practical situations. Also, one could investigate whether these ideas could be carried further using more powerful formal models, like some other forms of restricted PDAs, models where the communication channels have an infinite memory, or other formalisms that can represent timed events.

References

[1] A. Gargantini, "Conformance testing," in *Model-Based Testing of Reactive Systems: Advanced Lectures*, ser. Lecture Notes in Computer Science, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., vol. 3472. Springer-Verlag, 2005, pp. 87–111.

[2] D. P. Sidhu and T. Leung, "Formal methods for protocol testing: A detailed study," *IEEE Trans. Softw. Eng.*, vol. 15, no. 4, pp. 413–426, 1989.

[3] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing*, 2008, pp. 1–38.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, and others, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121213000563

[5] A. Simão and A. Petrenko, "Generating complete and finite test suite for ioco is it possible?" in *Ninth Workshop on Model-Based Testing (MBT 2014)*, 2014, pp. 56–70. [Online]. Available: http://arxiv.org/abs/1403.7261

[6] A. L. Bonifacio and A. V. Moura, "Testing asynchronous reactive systems: Beyond the ioco framework," *CLEI Electronic Journal*, vol. 24, no. 13, July 2021. [Online]. Available: https://doi.org/10.19153/cleiej.24.2.13

[7] R. Alur and P. Madhusudan, "Visibly pushdown languages," in *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '04. New York, NY, USA: ACM, 2004, pp. 202–211. [Online]. Available: http://doi.acm.org/10.1145/1007352.1007390

[8] C. Constant, B. Jeannet, and T. Jéron, "Automatic test generation from interprocedural specifications," in *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, ser. TestCom'07/FATES'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 41–57.

[9] Y. Li, Q. Zhang, and T. Reps, "On the complexity of bidirected interleaved dyck-reachability," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, jan 2021. [Online]. Available: https://doi.org/10.1145/3434340

[10] A. H. Kjelstrøm and A. Pavlogiannis, "The decidability and complexity of interleaved bidirected dyck reachability," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, jan 2022. [Online]. Available: https://doi.org/10.1145/3498673

[11] W. ACKERMANN, "Zum hilbertschen aufbau der reellen zahlen," *Mathematische Annalen*, vol. 99, pp. 118–133, 1928. [Online]. Available: http://eudml.org/doc/159248

[12] S. Chédor, T. Jéron, and C. Morvan, "Test generation from recursive tiles systems," in *Tests and Proofs*, A. D. Brucker and J. Julliand, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 99–114.

[13] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model checking pushdown systems," in *Proceedings of the 12th International Conference on Computer Aided Verification*, ser. CAV '00. Berlin, Heidelberg: Springer-Verlag, 2000, p. 232–247.

[14] A. Finkel, B. Willems, and P. Wolper, "A direct symbolic approach to model checking pushdown systems," in *INFINITY*, 1997, pp. 27–37.

[15] A. Okhotin and V. L. Selivanov, "Input-driven pushdown automata on well-nested infinite strings," in *Computer Science – Theory and Applications*, R. Santhanam and D. Musatov, Eds. Cham: Springer International Publishing, 2021, pp. 349–360.

[16] D. Chistikov, P. Martyugin, and M. Shirmohammadi, "Synchronizing automata over nested words," in *Foundations of Software Science and Computation Structures*, B. Jacobs and C. Löding, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 252–268.

[17] H. Fernau and P. Wolf, "Synchronization of deterministic visibly push-down automata," in *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020, December 14-18, 2020, BITS Pilani, K K Birla Goa Campus, Goa, India (Virtual Conference)*, ser. LIPIcs, N. Saxena and S. Simon, Eds., vol. 182. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 45:1–45:15. [Online]. Available: https://doi.org/10.4230/LIPIcs.FSTTCS.2020.45

[18] R. Senda, Y. Takata, and H. Seki, "Reactive synthesis from visibly register pushdown automata," in *Theoretical Aspects of Computing – ICTAC 2021*, A. Cerone and P. C. Ölveczky, Eds. Cham: Springer International Publishing, 2021, pp. 334–353.

[19] M. Sipser, *Introduction to the Theory of Computation.* International Thomson Publishing, 1996.