Noname manuscript No.
(will be inserted by the editor)

Model-based Testing for Concurrent Systems

Unfolding-based Test Selection

Hernán Ponce de León · Stefan Haar · Delphine Longuet

Received: date / Accepted: date

Abstract Model-based testing has mainly focused on models where concurrency is interpreted as interleaving (like the **ioco** theory for labeled transition systems), which may be too coarse when one wants concurrency to be preserved in the implementation. In order to test such concurrent systems, we choose to use Petri nets as specifications and define a concurrent conformance relation named **co-ioco**. We present a test generation algorithm based on Petri net unfolding able to build a complete test suite w.r.t our **co-ioco** conformance relation. In addition we propose several coverage criteria that allow to select finite prefixes of an unfolding in order to build manageable test suites.

1 Introduction

Model-based Testing. The aim of testing is to execute a software system, the *implementation*, on a set of input data selected so as to find discrepancies between actual behavior and intended behavior described by the *specification*. The testing process is usually decomposed into three phases: selection of relevant input data, called a *test suite*, among the possible inputs of the system; submission of this test suite to the implementation, its *execution*; and decision of the success or the failure of the test suite submission, known as the *oracle problem*.

Model-based testing requires a behavioral description of the system under test. One of the most popular formalisms studied in conformance testing is that of *input output labeled transition systems* (IOLTS). In this framework, the correctness (or conformance) relation the system under test (SUT) and its specification must verify is formalized by the **ioco** relation [23]. This relation has become a standard, and it is used as a basis in several testing theories for extended state-based models: restrictive transition systems [7, 16], symbolic transition systems [4, 13], timed automata [15], multi-port finite state machines [10].

Different Semantics for Concurrency. Systems composed of processes running in parallel are naturally modeled as a *network of finite automata*, a formal class of models that can be captured equivalently by *safe Petri nets*. Concurrency in a specification can arise for different reasons. First, two events may be physically localized on different processes, and thus be "naturally" independent of one another; this distribution is then part of the system construction. Second, the specification may not care about the order in which two actions are performed *on the same process*, and thus leave the choice of their ordering to the implementation. Depending on the nature of the concurrency specified in a given case, and thus on the intention of the specification, the *implementation relations* have to allow or disallow ordering of concurrent events. The kind of systems that we consider is of the first type, where concurrency comes from processes running in parallel. Therefore, we want concurrency of the specification to be preserved in the implementation.

We illustrate the need to preserve true concurrency (i.e. independence of actions) by an example coming from the field of security protocols. When designing a security protocol, an important property, named unlinkability, is to hide

This work was funded by the DIGITEO/DIM-LSC project TECSTES, convention DIGITEO Number 2011-052D - TECSTES.

This article is an extension of a paper published in ICTSS'13 [20].

Hernán Ponce de León · Stefan Haar
INRIA and LSV, École Normale Supérieure de Cachan and CNRS
61, avenue du Président Wilson 94235 CACHAN Cedex, France
E-mail: ponce@lsv.ens-cachan.fr, stefan.haar@inria.fr

Delphine Longuet
Univ Paris-Sud, LRI UMR8623, Orsay, F-91405
Bâtiment 650 Université Paris-Sud 11 91405 Orsay Cedex, France
E-mail: longuet@lri.fr

the information about the source of a message. An attacker that can identify messages as coming from the same source might use this information and thus threaten the privacy of the user. It has been shown that the security protocol of the French RFID e-passport is linkable, therefore anyone carrying a French e-passport can be physically traced [1]. While linkability can be interpreted as causality between messages, concurrency interpreted as interleavings cannot be used to model unlinkability. This property needs to be modeled using partial order semantics, and a correct implementation must preserve the independence between messages.

Model-based Testing of Concurrent Systems. Model-based testing of concurrent systems has been studied for a long time [8, 18, 21], however it is most of the time studied in the context of interleaving, or trace, semantics, which is known to suffer the state space explosion problem. To avoid this problem, non-interleaving models can be used for generation of test cases [9, 11]. Ulrich and König [24] propose a framework for testing concurrent systems specified by communicating labeled transition systems. The specification is translated into a Petri net, and a complete prefix of its unfolding is used to construct a *behavior machine*. The conformance relation proposed in [24] is a generalization of trace equivalence relation; their work does not include a test selection procedure, or how the choice of complete prefix impacts selection. Since our goal is to include *conflict* relations as well, we will use *event structures* and their properties.

Haar et al [25, 6] generalize the basic notions and techniques of I/O-sequence based conformance testing via a generalized I/O-automaton model where partially ordered patterns of input/output events are admitted as transition labels. However, these models still maintain a sequential automaton as the system's skeleton, and include synchronization constraints, e.g. all events in the course of a transition must be completed before any other transition can start.

Our Contribution. In order to enlarge the application domain, and add stronger benefits from concurrency modeling, we have introduced in [19] a conformance relation named **co-ioco**, as a generalization of **ioco**. This article is an extension of a paper published in ICTSS'13 [20]. In the original paper, we extend the work of [19] with a conformance relation where actions specified as concurrent must occur independently, on different processes, in any conformant implementation. Moreover we enlarge the conformance relation in order to test for refusals instead of considering the usual input-enabledness assumption on the implementation.

Besides the definition of a **co-ioco** conformance relation handling true concurrency, we define in [20] the notion of test case, we give sufficient properties for a test suite to be sound (not reject correct systems) and exhaustive (not accept incorrect systems), and we provide a test case generation algorithm that builds a complete (i.e. sound and exhaustive) test suite. We also propose a method to select a finite set of relevant test cases covering as many behaviors as possible (thus finding as many anomalies as possible). This selection method relies on the choice of a finite prefixe of the unfolding of the specification.

We extend here the work of [20] in two ways: we define new testing criteria based on different notions of prefixes of the unfolding; we propose a coverage measure that allows to compare these criteria with respect to the coverage they reach and their cost to reach it.

The testing approach we follow in this article is mostly theoretical: we study the testing problem from a centralized point of view, as a basis to the distributed testing problem. The global conformance relation we define is the relation we would like to be able to test in a distributed way (with local control and observation), and the global test cases are the basis for the construction of distributed tests.

Outline. The paper is organized as follows. Section 2 recalls basic notions about Petri nets, occurrence nets and labeled event structures. Section 3 introduces our testing hypotheses and our **co-ioco** conformance relation. In Section 4, we define the notion of complete test suite, give sufficient conditions for a test suite to be complete and an algorithm producing such a test suite. In Section 5, we define different selection criteria and adapt the complete finite prefix algorithm of [3] to build a sound test suite satisfying a given criterion. A comparison of these criteria concludes.

2 I/O Petri Nets and their Semantics

We choose to use *Petri nets* as specifications to have explicit concurrency. The semantics associated to a Petri net is given by its unfolding to an *occurrence net*, which can also be seen as an *event structure*. We will present both notions since we use them in different contexts in the following. The execution traces for this semantics are not sequences but *partial orders*, which keep concurrency explicit. We recall here these basic notions.

I/O Petri Nets. A *net* is a tuple $N = (P, T, F)$ where *(i)* $P \neq \emptyset$ is a set of *places*, *(ii)* $T \neq \emptyset$ is a set of *transitions* such that $P \cap T = \emptyset$, *(iii)* $F \subseteq (P \times T) \cup (T \times P)$ is a set of *flow arcs*. A *marking* is a multiset M of places, i.e. a map $M : P \to \mathbb{N}$. A *Petri net* is a tuple $\mathcal{N} = (P, T, F, M_0)$, where *(i)* (P, T, F) is a finite net, and *(ii)* $M_0 : P \to \mathbb{N}$ is an *initial marking*. Elements of $P \cup T$ are called the *nodes* of $\mathcal{N}$. For a transition $t \in T$, we call ${}^\bullet t = \{p \mid (p, t) \in F\}$ the *preset* of t, and $t^\bullet = \{p \mid (t, p) \in F\}$ the *postset* of t. In figures, we represent as usual places by empty circles, transitions by squares, F by arrows, and the marking

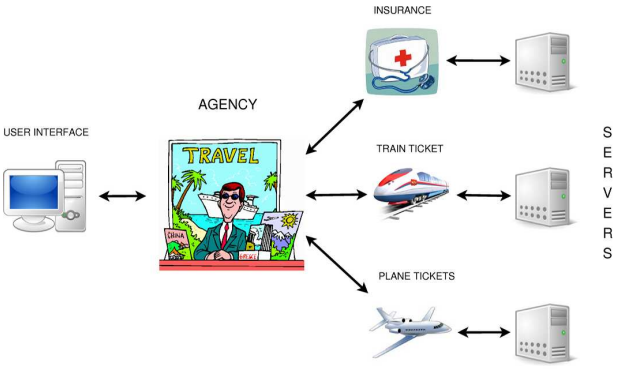


Fig. 1 A travel agency example

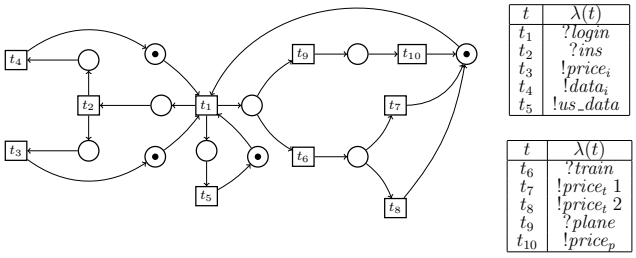


Fig. 2 I/O Petri net of the travel agency

of a place p by black tokens in p. A transition t is *enabled* in marking M, written $M \xrightarrow{t}$, if $\forall p \in {}^\bullet t,\ M(p) > 0$. This enabled transition can *fire*, resulting in a new marking $M' = M - {}^\bullet t + t^\bullet$. This firing relation is denoted by $M \xrightarrow{t} M'$. A marking M is *reachable* from M_0 if there exists a *firing sequence*, i.e. transitions $t_0 \dots t_n$ such that $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \dots \xrightarrow{t_n} M$. The set of markings reachable from M_0 (in $\mathcal{N}$) is denoted $\mathbf{R}_\mathcal{N}(M_0)$ (we drop the subscript referring to $\mathcal{N}$ when it is clear from the context). A Petri net $\mathcal{N} = (P, T, F, M_0)$ is *(1-)safe* iff for all reachable markings $M \in \mathbf{R}(M_0)$, $M(p) \in \{0, 1\}$ for all $p \in P$.

Let $\mathcal{I}$ and $\mathcal{O}$ be two disjoint non-empty sets of *input* and *output* labels, respectively. For a net $N = (P, T, F)$, a mapping $\lambda : T \to (\mathcal{I} \uplus \mathcal{O})$ is called an *I/O-labeling*. Denote by $T^\mathcal{I}$ and $T^\mathcal{O}$ the input and output transition sets, respectively; that is, $T^\mathcal{I} \triangleq \lambda^{-1}(\mathcal{I})$ and $T^\mathcal{O} \triangleq \lambda^{-1}(\mathcal{O})$. An *I/O Petri net* is a pair $\Sigma = (\mathcal{N}, \lambda)$, where $\mathcal{N} = (P, T, F, M_0)$ is a 1-safe Petri net and $\lambda : T \to (\mathcal{I} \uplus \mathcal{O})$ an I/O-labeling. Σ is called *deterministically labeled* iff no two transitions with the same label are simultaneously enabled, i.e. for all $t_1, t_2 \in T$ and $M \in \mathbf{R}(M_0)$:

$$(M \xrightarrow{t_1} \ \wedge\ M \xrightarrow{t_2} \ \wedge\ \lambda(t_1) = \lambda(t_2)) \Rightarrow t_1 = t_2$$

Note that 1-safeness of the Petri net is not sufficient for guaranteeing deterministic labeling. Deterministic labeling ensures that the system's behavior is locally distinguishable through labels, either through distinct inputs or through observation of different outputs.

Example 1 Fig. 2 shows a schematic travel agency whose behavior can be formally specified by the I/O Petri net presented in Fig. 2, where ? denotes input actions and ! output ones. In this system, once the user has logged in ($?login$), some data is sent to the server ($!us\_data$) and he can choose an insurance ($?ins$) and a train ticket ($?train$) or a plane ticket ($?plane$). If a plane ticket is chosen, its price is sent to the user ($!price_p$). If a train ticket is selected, two kind of prices can be proposed: a first class ($!price_t\ 1$) or a second class one ($!price_t\ 2$). The insurance choice is followed by its price ($!price_i$) and some extra data that is sent to the user ($!data_i$).

When testing reactive systems, we need to differentiate situations where the system can still produce some outputs and those where the system can not evolve without an input from the environment. Such situations are captured by the notion of *quiescence* [22]. A marking is said quiescent if it does not enable output transitions, i.e. $M \xrightarrow{t}$ implies $t \in T^\mathcal{I}$. The observation of quiescence is usually instrumented by timers. Jard and Jéron [12] present three different kinds of quiescence: *output quiescence* when the system is waiting for an input from the environment, *deadlock* when the system can not evolve anymore, and *livelock* when the system diverges by an infinite sequence of silent actions.

Occurrence Nets and Unfoldings. Occurrence nets can be seen as Petri nets[1] with a special acyclic structure that highlights *conflict* between transitions that compete for resources. Formally, let $N = (P, T, F)$ be a net, $<$ the transitive closure of F, and $\leqslant$ the reflexive closure of $<$. We say that transitions t_1 and t_2 are in *structural conflict*, written $t_1 \#^\omega t_2$, if and only if $t_1 \neq t_2$ and ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. *Conflict* is inherited along $<$, that is, the conflict relation $\#$ between transitions $a, b \in T$ is given by

$$a \# b \Leftrightarrow \exists t_a, t_b \in T : t_a \#^\omega t_b \wedge t_a \leqslant a \wedge t_b \leqslant b$$

Finally, the *concurrency relation* **co** holds between nodes $a, b \in P \cup T$ that are neither ordered nor in conflict, i.e. a **co** $b \Leftrightarrow \neg(a \leqslant b) \ \wedge\ \neg(a \# b) \ \wedge\ \neg(b < a)$.

Definition 1 A net $ON = (B, E, G)$ is an *occurrence net* if and only if

1. $\leqslant$ is a partial order;
2. for all $b \in B$, $|{}^\bullet b| \in \{0, 1\}$;
3. for all $x \in B \cup E$, the set $[x] = \{y \in E \mid y \leqslant x\}$ is finite;
4. no self-conflict, i.e. there is no $x \in B \cup E$ such that $x \# x$;
5. $\perp \in E$ is the only $\leq$-minimal node (event $\perp$ creates the initial conditions)

[1] when one allows Petri nets to be *infinite*

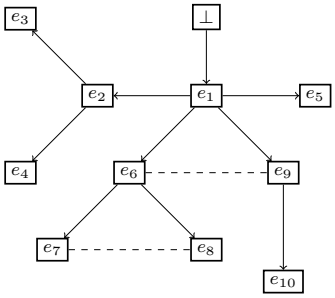
Call the elements of E *events*, those of B *conditions*. A set of conditions is a co-set if its elements are pairwise in **co** relation. A maximal co-set with respect to set inclusion is called a *cut*. An ON can also be given as a tuple $(B, E\backslash\{\bot\}, F, cut_0)$, where $cut_0 = \bot^\bullet$ is the set of minimal conditions. Given an occurrence net $ON = (B, E, G)$, every $\leq$-closed set of events $E' \subseteq E$ induces a *prefix* with conditions $({}^\bullet E \cup E^\bullet)$.

Occurrence nets are the mathematical form of the *partial order unfolding semantics* [3]. A *branching process* of a 1-safe Petri net $\mathcal{N} = (N, M_0)$ is given by a pair $\Phi = (ON, \varphi)$, where $ON = (B, E, G)$ is an occurrence net, and $\varphi : B \cup E \to P \cup T$ is such that:

1. it is a homomorphism from ON to N, i.e.
   - $\varphi(B) \subseteq P$ and $\varphi(E) \subseteq T$, and
   - for every $e \in E$, the restriction of φ to ${}^\bullet e$ is a bijection between the set ${}^\bullet e$ in ON and the set ${}^\bullet\varphi(e)$ in N, and similarly for $e^\bullet$ and $\varphi(e)^\bullet$;
2. the restriction of φ to cut_0 is a bijection from cut_0 to M_0; and
3. for every $e_1, e_2 \in E$, if ${}^\bullet e_1 = {}^\bullet e_2$ and $\varphi(e_1) = \varphi(e_2)$ then $e_1 = e_2$.

The unique (up to isomorphism) maximal (w.r.t prefixes) branching process $\mathcal{U} = (ON_\mathcal{U}, \varphi_\mathcal{U})$ of $\mathcal{N}$ is called the *unfolding* of $\mathcal{N}$.

I/O Labeled Event Structures. Occurrence nets give rise to event structures in the sense of Winskel et al [17]; as usual, we will use both the event structure and the occurrence net formalism, whichever is more convenient. An *input/output labeled event structure (IOLES)* over an alphabet $L = \mathcal{I} \uplus \mathcal{O}$ is a 4-tuple $\mathcal{E} = (E, \leq, \#, \lambda)$ where *(i)* E is a set of events, *(ii)* $\leq \subseteq E \times E$ is a partial order (called *causality*) satisfying the property of *finite causes*, i.e. $\forall e \in E : |\{e' \in E \mid e' \leq e\}| < \infty$, *(iii)* $\# \subseteq E \times E$ is an irreflexive symmetric relation (called *conflict*) satisfying the property of *conflict heredity*, i.e. $\forall e, e', e'' \in E : e \# e' \wedge e' \leq e'' \Rightarrow e \# e''$, *(iv)* $\lambda : E \to (\mathcal{I} \uplus \mathcal{O})$ is a labeling mapping. In addition, we assume every IOLES $\mathcal{E}$ has a unique minimal event $\bot_\mathcal{E}$. We denote the class of all input/output labeled event structures over L by $\mathcal{IOLES}(L)$. Given event e, its *local configuration* is $[e] \triangleq \{e' \in E \mid e' \leq e\}$, and its set of *causal predecessors* is $\langle e\rangle \triangleq [e]\backslash\{e\}$. Two events $e, e' \in E$ are said to be concurrent (e **co** e') iff neither $e \leq e'$ nor $e' \leq e$ nor $e \# e'$ hold; $e, e' \in E$ are in *immediate conflict* $(e_1 \#^\mu e_2)$ iff $[e_1] \times [e_2] \cap \# = \{(e_1, e_2)\}$. A *configuration* of an IOLES is a non-empty set $C \subseteq E$ that is *(i) causally closed*, i.e. $e \in C$ implies $[e] \subseteq C$, and *(ii) conflict-free*, i.e. $e \in C$ and $e \# e'$ imply $e' \notin C$. Note that we define, for technical convenience, all configurations to be non-empty; the initial configuration of $\mathcal{E}$, containing only $\bot_\mathcal{E}$ and denoted by $\bot_\mathcal{E}$, is contained in every configuration of $\mathcal{E}$. We denote the set of all the configurations of $\mathcal{E}$ by $\mathcal{C}(\mathcal{E})$ and the set of maximal configurations (those that can not be extended) by $\Omega(\mathcal{E})$.



e	$\varphi(e)$	$\lambda(e)$
e_1	t_1	$?login$
e_2	t_2	$?ins$
e_3	t_3	$!price_i$
e_4	t_4	$!data_i$
e_5	t_5	$!us\_data$
e_6	t_6	$?train$
e_7	t_7	$!price_t\ 1$
e_8	t_8	$!price_t\ 2$
e_9	t_9	$?plane$
e_{10}	t_{10}	$!price_p$

Fig. 3 Part of the unfolding of the PN from Fig. 2 represented as an IOLES. Causality is represented by arrows and immediate conflict by dashed lines.

Example 2 In the travel agency example, as can be seen in Fig. 3, the data cannot be sent before the user logged in ($?login \leq !us\_data$) and the selections for a ticket and an insurance can be done concurrently ($?train$ **co** $?ins$), but only one ticket can be chosen ($?train \ \# \ ?plane$). From the conflict heredity property, only one ticket price is produced ($!price_t\ 1 \ \# \ !price_p$ and $!price_t\ 2 \ \# \ !price_p$). A transition from the net is usually represented by several events in its unfolding: other instances of t_1 ($?login$) can be added to the unfolding causally depending on e_3, e_4, e_5 and either e_7, e_8 or e_{10} as it is shown in Fig 8.

Labeled Partial Orders. We are interested in testing distributed systems where concurrent actions occur in different processes of the system. For this reason, we want to keep concurrency explicit, i.e. implementations do not impose any order of execution between concurrent events. Labeled partial orders can then be used to represent executions of such systems. A *labeled partial order* (lpo) is a tuple $lpo = (E, \leq, \lambda)$ where E is a set of events, $\leq$ is a reflexive, antisymmetric, and transitive relation, and $\lambda : E \to L$ is a labeling mapping to a fix alphabet L. We denote the class of all labeled partial orders over L by $\mathcal{LPO}(L)$. Consider $lpo_1 = (E_1, \leq_1, \lambda_1)$ and $lpo_2 = (E_2, \leq_2, \lambda_2) \in \mathcal{LPO}(L)$. A bijective function $f : E_1 \to E_2$ is an isomorphism between lpo_1 and lpo_2 iff *(i)* $\forall e, e' \in E_1 : e \leq_1 e' \Leftrightarrow f(e) \leq_2 f(e')$ and *(ii)* $\forall e \in E_1 : \lambda_1(e) = \lambda_2(f(e))$. Two labeled partial orders lpo_1 and lpo_2 are isomorphic if there exists an isomorphism between them. A *partially ordered multiset* (pomset) is an isomorphism class of lpos. We will represent such a class by one of its objects. Denote the class of all non empty pomsets over L by $\mathcal{POMSET}(L)$. The evolution of the system is captured by the following definition: pomsets are observations.

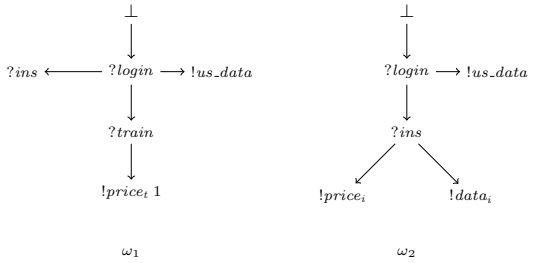


Fig. 4 Traces of the travel agency.

Definition 2 For $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$, $\omega \in \mathcal{POMSET}(L)$ and $C, C' \in \mathcal{C}(\mathcal{E})$, define

$$C \overset{\omega}{\Longrightarrow} C' \triangleq \exists lpo = (E_\omega, \leq_\omega, \lambda_\omega) \in \omega : E_\omega \subseteq E \backslash C, \; C' = C \cup E_\omega, \leq \cap (E_\omega \times E_\omega) = \leq_\omega \text{ and } \lambda_{|E_\omega} = \lambda_\omega$$

$$C \overset{\omega}{\Longrightarrow} \quad \triangleq \exists C' : C \overset{\omega}{\Longrightarrow} C'$$

Example 3 Consider Fig. 2. Both ω_1 and ω_2 are partial orders that respect the structure of the unfolding in Fig. 3. Therefore we have $\perp \overset{\omega_1}{\Longrightarrow} \{\perp, e_1, e_2, e_5, e_6, e_7\}$ and $\perp \overset{\omega_2}{\Longrightarrow} \{\perp, e_1, e_2, e_3, e_4, e_5\}$.

Remark 1 When the system is composed of a single process, every configuration C generates a cut $C^\bullet = \{q\}$ in the unfolding where q represents the current state of the process. In this case, Definition 2 and the definition of $\Longrightarrow$ for LTS presented in [23] coincide.

We can now define the notions of traces and of configurations reachable from a given configuration by an observation. Our notion of traces is similar to the one of Ulrich and König [24].

Definition 3 For $\mathcal{E} \in \mathcal{IOLES}(L)$, $\omega \in \mathcal{POMSET}(L)$, $C, C' \in \mathcal{C}(\mathcal{E})$, define

$$\text{traces}(\mathcal{E}) \triangleq \{\omega \in \mathcal{POMSET}(L) \mid \perp_\mathcal{E} \overset{\omega}{\Longrightarrow}\}$$

$$C \textbf{ after } \omega \triangleq \{C' \mid C \overset{\omega}{\Longrightarrow} C'\}$$

Remark 2 Note that for deterministically labeled I/O Petri nets, every configuration of the corresponding IOLES cannot enable two equally labeled events, i.e. the IOLES is deterministic and the set of reachable configurations is a singleton.

3 Testing Framework for IOPNs

Testing Hypotheses. We assume that the specification of the system under test is given as a 1-safe and deterministically labeled I/O Petri net $\Sigma = (\mathcal{N}, \lambda)$ over alphabet $L = \mathcal{I} \uplus \mathcal{O}$ of input and output labels. To be able to test an implementation against such a specification, we make a set of testing assumptions. First of all, we make the usual testing assumption that the behavior of the SUT itself can be modeled by a 1-safe I/O Petri net over the same alphabet of labels. We also assume as usual that the specification does not contain cycles of outputs actions, so that the number of expected outputs after a given trace is finite.

Assumption 1 *The net $\mathcal{N}$ has no cycle containing only output transitions.*

Third, in order to allow the observation of both the outputs produced by the system and the inputs it can accept, markings where conflicting inputs and outputs are enabled should not be reachable. As a matter of fact, if conflicting input and output are enabled in a given marking, once the output is produced, the input is not enabled anymore, and vice versa. Such markings prevent from observing the inputs enabled in a given configuration, which we will see is one of the key points of our conformance relation. For this reason, we restrict the form of the nets we consider via the following assumption on the unfolding[2]:

Assumption 2 *The unfolding of the net $\mathcal{N}$ has no immediate conflict between input and output events, i.e. $\forall e_1 \in E^\mathcal{I}, e_2 \in E^\mathcal{O} : \neg(e_1 \#^\mu e_2)$.*

Conformance Relation. A formal testing framework relies on the definition of a conformance relation to be satisfied by the SUT and its specification. In the LTS framework, the **ioco** conformance relation compares the outputs and blockings in the implementation after a trace of the specification to the outputs and blockings authorised after this trace in the specification. Classically, the produced outputs of the system under test are elements of $\mathcal{O}$ (single actions) and blockings are observable by a special action $\delta \notin L$ which represents the expiration of a timer.

By contrast, in partial order semantics, we need any set of outputs to be entirely produced by the system under test before we send a new input; this is necessary to detect outputs depending on extra inputs. Suppose two concurrent outputs o_1 and o_2 depending on input i_1 and another input i_2 depending on both outputs. Clearly, an implementation that accepts i_2 before o_2 should not be considered as correct, but if i_2 is sent too early to the system, we may not know if the occurrence of o_2 depends or not on i_2. For this reason we define the expected outputs from a configuration C as the pomset of outputs leading to a quiescent configuration. Such a configuration always exists, and must be finite by Assumption 1.

The notion of quiescence is inherited from nets, i.e. a configuration C is quiescent if and only if $C \overset{\omega}{\Longrightarrow}$ implies

[2] Gaudel et al [16] assume a similar property called *IO-exclusiveness.*

$\omega \notin \mathcal{POMSET}(\mathcal{O})$. We assume as usual that quiescence is observable by a special δ action, i.e. C is quiescent iff $C \stackrel{\delta}{\Longrightarrow}$.

Definition 4 For $\mathcal{E} \in \mathcal{IOLES}(L),\ C \in \mathcal{C}(\mathcal{E})$, the outputs produced by C are

$$\text{out}_{\mathcal{E}}(C) \triangleq \{!\omega \in \mathcal{POMSET}(\mathcal{O}) \mid C \stackrel{!\omega}{\Longrightarrow} C' \ \wedge\ C' \stackrel{\delta}{\Longrightarrow}\} \cup \{\delta \mid C \stackrel{\delta}{\Longrightarrow}\}$$

The **ioco** theory assumes the input enabledness of the implementation [23], i.e. in any state of the implementation, every input action is enabled. This assumption is made to ensure that no blocking can occur during the execution of the test until its end and the emission of a verdict. However, as explained by Heerink [7] and Lestiennes and Gaudel [16] even if many realistic systems can be modeled with such an assumption, there remains a significant portion of realistic systems that can not be modeled as such. In order to overcome these difficulties, Lestiennes and Gaudel enrich the system model by refused transitions and a set of possible actions is defined in each state. Any possible input in a given state of the specification should be possible in a correct implementation.

Definition 5 For $\mathcal{E} \in \mathcal{IOLES}(L)$ and $C \in \mathcal{C}(\mathcal{E})$, the possible inputs in C are

$$\text{poss}_{\mathcal{E}}(C) \triangleq \{?\omega \in \mathcal{POMSET}(\mathcal{I}) \mid C \stackrel{?\omega}{\Longrightarrow}\}$$

Our **co-ioco** conformance relation for labeled event structures can be informally described as follows. The behavior of a correct **co-ioco** implementation after some observations (obtained from the specification) should respect the following restrictions: (1) the outputs produced by the implementation should be specified; (2) if a quiescent configuration is reached, this should also be the case in the specification; (3) any time an input is possible in the specification, this should also be the case in the implementation. These restrictions are formalized by the following conformance relation[3].

Definition 6 Let $\mathcal{E}_i, \mathcal{E}_s \in \mathcal{IOLES}(L)$, then

$$\mathcal{E}_i \textbf{ co-ioco } \mathcal{E}_s \Leftrightarrow \forall \omega \in \text{traces}(\mathcal{E}_s) : \text{poss}_s(\bot \textbf{ after } \omega) \subseteq \text{poss}_i(\bot \textbf{ after } \omega)\ \ \text{out}_i(\bot \textbf{ after } \omega) \subseteq \text{out}_s(\bot \textbf{ after } \omega)$$

When several outputs in conflicts are possible, our conformance relation allows implementations where at least one of them is implemented. Extra inputs are allowed in any configuration, but extra outputs, extra quiescence and extra causality or concurrency are forbidden.

[3] As we consider only deterministically labeled nets, by Remark 2, $(\bot$ **after** $\omega)$ is always a singleton.

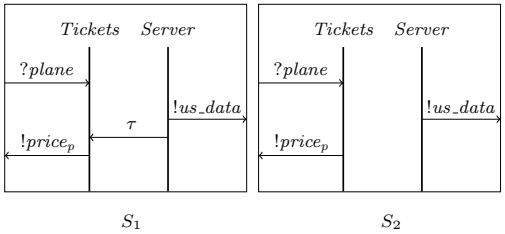


Fig. 5 Message sequence charts showing two implementations of concurrency.

Consider Fig. 5. In the **ioco** theory where concurrency is interpreted as interleaving, the concurrency between outputs $!price_p$ and $!us\_data$ of system S_2 would be described allowing either $!price_p$ before $!us\_data$ or $!us\_data$ before $!price_p$. S_1 would be a correct implementation w.r.t **ioco** because one of the two possible orders between the outputs is observed (the τ action is unobservable), even if process P_2 interferes in the behavior of process P_1 ($!price_p$ depends on $!us\_data$). We want to prevent implementations like S_1 introducing extra dependency between events specified as concurrent. Therefore actions specified as concurrent must be implemented as such, meaning that they must occur on different processes and must be independent from each other. The **co-ioco** conformance relation detects this kind of non conformant implementation. However, when there is no concurrency in the system (the system is composed of a single process), by Remark 1 we can conclude that **ioco** and **co-ioco** coincide.

4 Complete Test Suites

A global test case is a specification of the tester's behavior during an experiment carried out on the SUT. It must be controllable, i.e. the tester must not have choices to make during the execution of the test. That is, tests must be deterministic, and at any stage, the next input to be proposed by the tester must be unique, i.e. there are no immediate conflicts between inputs. Finally, we require the experiment to terminate, i.e. the resulting event structure must be finite.

Definition 7 A global test case is a finite deterministic IOLES $\mathcal{E}_t = (E_t, \leq_t, \#_t, \lambda_t)$ where $(E_t^{\mathcal{I}} \times E_t^{\mathcal{I}}) \cap \#_t^{\mu} = \emptyset$. A test suite is a set of test cases.

As global test cases are defined as IOLES, concurrency of the specification is preserved.

4.1 Test Execution

The *success* of a test is determined by the verdict associated to the result of its *execution* on the system, *pass* or *fail*, the pass verdict meaning that the result of the test is consistent with the specification according to the conformance relation.

The interaction between two systems is usually formalized by their parallel composition. This composition assumes that both systems are always prepared to accept an output that the other may produce. In the sequential setting, it is assumed that the implementation accepts any input the tester can propose (input enableness of the implementation). Analogously, the tester should be able to synchronize with any output the implementation may produce. Constructing an event structure having such a property is almost impossible due to the fact that it should not only accept any output, but also all the possible ways such an output could happen (concurrently/sequentially with other outputs). In addition, the parallel composition of nets [26] does not preserve concurrency. We propose another approach to formalize the interaction between the implementation and a test case.

Deadlocks of the parallel composition are used to give verdicts about the test run in the sequential framework. Such deadlocks are produced in the following situations: (1) the implementation proposes an output or δ action that the test case can not accept, (2) the test case proposes an input that the implementation can not accept, or (3) the test case has nothing else to propose (it deadlocks). The first two situations lead to a fail verdict and the last one to a pass one. For having such verdicts, we will define the notion of blocking in the test execution.

After observing a trace, the test execution can block because of an output or δ action the implementation produces. This happens if after such an observation the test case can not accept that action or if the reached configuration is not quiescent, i.e. the implementation produces an output that the test case is not prepared to accept.

Definition 8 Let $i, t \in \mathcal{IOLES}(L)$ be an implementation and a test case respectively and $\omega \in \mathcal{POMSET}(L)$, we have $\textbf{blocks}_{\mathcal{O}}(i,t,\omega) \Leftrightarrow \exists x \in \text{out}_i(\bot \textbf{ after } \omega) : x \notin \text{out}_t(\bot \textbf{ after } \omega)$ with $x \in \mathcal{POMSET}(\mathcal{O}) \cup \{\delta\}$.

The other blocking situation happens when the test case can propose an input that the implementation is not prepared to accept.

Definition 9 Let $i, t \in \mathcal{IOLES}(L)$ be an implementation and a test case respectively and $\omega \in \mathcal{POMSET}(L)$, we have $\textbf{blocks}_{\mathcal{I}}(i,t,\omega) \Leftrightarrow \exists ?\omega \in \text{poss}_t(\bot \textbf{ after } \omega) : ?\omega \notin \text{poss}_i(\bot \textbf{ after } \omega)$.

We can now define the verdict of the executions of a set a test cases on an implementation based on the notions of blockings.

Definition 10 Let i be an implementation, and T a test suite, we have i **fails** $T \Leftrightarrow \exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i,t,\omega) \vee \textbf{blocks}_{\mathcal{I}}(i,t,\omega)$. If the implementation does not fail the test suite, it passes it.

4.2 Completeness of the Test Suite

We expect our test suite to be *sound*, i.e. if the implementation fails the test, then it does not conform to the specification. A test suite is *exhaustive* iff it contains, for every non conforming implementation, a test that detects it. The existence of a *complete* (sound and exhaustive) test suite ensures *testability* of the conformance relation, since success of the SUT under such a test suite proves the conformance of the SUT.

Definition 11 Let s be a specification and T a test suite, then

T is sound	$\triangleq \forall i : i$ **fails** T	implies	$\neg(i$ **co-ioco** $s)$
T is exhaustive	$\triangleq \forall i : i$ **fails** T	if	$\neg(i$ **co-ioco** $s)$
T is complete	$\triangleq \forall i : i$ **fails** T	iff	$\neg(i$ **co-ioco** $s)$

The following theorem gives sufficient conditions for having a sound test suite: each test must produce only traces of the specification, and preserve all possible outputs for each such trace.

Theorem 1 *Let $\mathcal{E}_s \in \mathcal{IOLES}(L)$ and T a test suite such that*

1. $\forall \mathcal{E}_t \in T : \textit{traces}(\mathcal{E}_t) \subseteq \textit{traces}(\mathcal{E}_s)$
2. $\forall \mathcal{E}_t \in T, \omega \in \textit{traces}(\mathcal{E}_t) : \textit{out}_s(\bot \textbf{ after } \omega) \subseteq \textit{out}_t(\bot \textbf{ after } \omega)$

*then T is sound for $\mathcal{E}_s$ w.r.t **co-ioco**.*

Assumption 1. in the theorem above guarantees that any possible input in the test case is a possible input of the specification, i.e. $\text{poss}_t(\bot \textbf{ after } \omega) \subseteq \text{poss}_s(\bot \textbf{ after } \omega)$.

Proof. T is sound for s w.r.t. **co-ioco** iff for every implementation i that fails the test suite, we have that it does not conform to the specification. We assume i **fails** T and by Definition 10 we have:

$$\exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i,t,\omega) \vee \textbf{blocks}_{\mathcal{I}}(i,t,\omega)$$

and at least one of the following cases holds:

1. the test execution blocks after ω because of an output produced by the implementation:

$$\begin{array}{ll} & \exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 8 } *\} \\ & \exists t \in T, \omega \in \text{traces}(t) : \\ & \text{out}_i(\bot \textbf{ after } \omega) \not\subseteq \text{out}_t(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Assumptions 1. and 2. } *\} \\ & \exists \omega \in \text{traces}(s) : \\ & \text{out}_i(\bot \textbf{ after } \omega) \not\subseteq \text{out}_s(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 6 } *\} \\ & \neg(i \textbf{ co-ioco } s) \end{array}$$

2. the test execution blocks after ω because of an input proposed by the test case:

$$\begin{array}{ll} & \exists \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{I}}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 9 } *\} \\ & \exists \omega \in \text{traces}(t) : \\ & \text{poss}_t(\bot \textbf{ after } \omega) \not\subseteq \text{poss}_i(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Assumption 1. twice } *\} \\ & \exists \omega \in \text{traces}(s) : \\ & \text{poss}_s(\bot \textbf{ after } \omega) \not\subseteq \text{poss}_i(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 6 } *\} \\ & \neg(i \textbf{ co-ioco } s) \end{array}$$

□

A test suite is exhaustive if each trace of the specification appears in at least one test case.

Theorem 2 *Let $\mathcal{E}_s \in \mathcal{IOLECS}(L)$ and T a test suite such that $\forall \omega \in$ traces$(\mathcal{E}_s), \exists \mathcal{E}_t \in T : \omega \in$ traces$(\mathcal{E}_t)$, then T is exhaustive for $\mathcal{E}_s$ w.r.t* ***co-ioco****.*

Proof. We need to prove that if i does not conform to s then i **fails** T. We assume $\neg(i \textbf{ co-ioco } s)$, then at least one of the following two cases holds:

1. The implementation does not conform to the specification because an output produced by the implementation is not specified:

$$\begin{array}{ll} & \exists \omega \in \text{traces}(s) : \\ & \exists x \in \text{out}_i(\bot \textbf{ after } \omega) : x \notin \text{out}_s(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ by the assumption, we choose } t \text{ such that} \\ & \omega \in \text{traces}(t) \text{ and } x \notin \text{out}_t(\bot \textbf{ after } \omega) \; *\} \\ & \exists t \in T, \omega \in \text{traces}(t) : \\ & \exists x \in \text{out}_i(\bot \textbf{ after } \omega) : x \notin \text{out}_t(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 8 } *\} \\ & \exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 10 } *\} \\ & i \textbf{ fails } T \end{array}$$

2. The implementation does not conform to the specification because an input from the specification is not possible in the implementation:

$$\begin{array}{ll} & \exists \omega \in \text{traces}(s) : \exists ?\omega \in \text{poss}_s(\bot \textbf{ after } \omega) : \\ & ?\omega \notin \text{poss}_i(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 5 } *\} \\ & \exists (\omega \cdot ?\omega) \in \text{traces}(s) : ?\omega \notin \text{poss}_i(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ by the assumption, we choose } t \text{ such that} \\ & (\omega \cdot ?\omega) \in \text{traces}(t) \; *\} \\ & \exists t \in T : (\omega \cdot ?\omega) \in \text{traces}(t) \text{ and} \\ & ?\omega \notin \text{poss}_i(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 5 } *\} \\ & \exists t \in T, \omega \in \text{traces}(t), ?\omega \in \text{poss}_t(\bot \textbf{ after } \omega) : \\ & ?\omega \notin \text{poss}_i(\bot \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 9 } *\} \\ & \exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{I}}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 10 } *\} \\ & i \textbf{ fails } T \end{array}$$

□

4.3 Test Suite Generation

The algorithm below builds a global test case from an IOLES by resolving immediate conflicts between inputs, while accepting several branches in case of conflict between outputs (note that "mixed" immediate conflicts between inputs and outputs have been ruled out by Assumption 2). At the end of the algorithm, all such conflicts have been resolved in one way, following one fixed strategy of resolution of immediate input conflicts; the resulting object, the test case, is thus one branching prefix of the IOLES. In order to cover the other branches, the algorithm must be run several times with *different* conflict resolution schemes, to obtain a test suite that represents every possible event in at least one test case. Each such scheme can be represented as a linearization of the causality relation that specifies in which order the events are selected by the algorithm. By the above, we need to be sure that the collection of linearizations that we use considers all resolutions of immediate input conflict, i.e. is rich enough such that there is a pair of linearizations that reverses the order in a given immediate input conflict.

Definition 12 Fix $\mathcal{E} \in \mathcal{IOLECS}(L)$, and let $\mathcal{L}$ be a set of linearizations of $\leq$. Then $\mathcal{L}$ is an *immediate input conflict saturated* set, or *iics* set, for $\mathcal{E}$ iff for all $e_1, e_2 \in E^{\mathcal{I}}$ such that $e_1 \#^{\mu} e_2$, there exist $\mathcal{R}_1, \mathcal{R}_2 \in \mathcal{L}$ with $\forall e \in [e_1] : e\mathcal{R}_1 e_2$ and $\forall e \in [e_2] : e\mathcal{R}_2 e_1$.

Algorithm 1 Test Case Construction

Require: A finite and deterministically labeled $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLECS}(L)$ such that $\forall e \in E^{\mathcal{I}}, e' \in E^{\mathcal{O}} : \neg(e \#^{\mu} e')$ and a linearization $\mathcal{R}$ of $\leq$
Ensure: A test case $\mathcal{E}_t$ such that
$\forall \omega \in \text{traces}(\mathcal{E}_t) : \text{out}_{\mathcal{E}_t}(\bot \textbf{ after } \omega) = \text{out}_{\mathcal{E}}(\bot \textbf{ after } \omega)$

```
E_t := ∅
E_temp := E
while E_temp ≠ ∅ do
    e_m := min_R(E_temp)
    E_temp := E_temp \ {e_m}
    if ({e_m} × E_t^I) ∩ #^μ = ∅ ∧ ⟨e_m⟩ ⊆ E_t then
        E_t := E_t ∪ {e_m}
    end if
end while
≤_t := ≤ ∩ (E_t × E_t)
#_t := # ∩ (E_t × E_t)
λ_t := λ_|E_t
return E_t = (E_t, ≤_t, #_t, λ_t)
```

Proposition 1 *Let $\mathcal{L}$ be an iics set for $\mathcal{E}$, and T the test suite obtained using Algorithm 1 with $\mathcal{L}$. Then every event $e \in E$ is represented by at least one test case $\mathcal{E}_t \in T$.*

Proof Let T be the test suite obtained by the algorithm and $\mathcal{L}$ and suppose e is not represented by any test case in T. We have then that for every $\mathcal{E}_t \in T$ either *(i)* $e \in E^{\mathcal{I}}$ and $\{e\} \times E^{\mathcal{I}}_t \cap \#^{\mu} \neq \emptyset$ or *(ii)* $[e] \not\subseteq E_t$. If *(i)*, we have that there exists $e' \in E^{\mathcal{I}}_t$ such that $e \#^{\mu} e'$ and $e' \mathcal{R}_1 e$ (where $\mathcal{R}_1$ is the linearization used to build $\mathcal{E}_t$). By Definition 12 we know there exist $\mathcal{R}_2 \in \mathcal{L}$ such that $\forall e'' \in [e] : e'' \mathcal{R}_2 e'$ and then we can use $\mathcal{R}_2$ to construct $\mathcal{E}'_t \in T$ such that e is represented by $\mathcal{E}'_t$ which leads to a contradiction. If *(ii)*, then there exists $e' \in [e]$ such that $\{e'\} \times E^{\mathcal{I}}_t \cap \#^{\mu} \neq \emptyset$ and the analysis is analogous to the one in *(i)*. □

Note that the size of $\mathcal{L}$ and hence of T can be bounded by the number of input events in immediate conflict, i.e. $|T| \leq 2^{\mathcal{K}}$, where $\mathcal{K} = |\#^{\mu} \cap (E^{\mathcal{I}} \times E^{\mathcal{I}})|$. Note that in the case where several input events are two by two in immediate conflict, we need fewer test cases than one per pair. For example if $e_1 \#^{\mu} e_2, e_2 \#^{\mu} e_3$ and $e_3 \#^{\mu} e_1$, we only need three linearizations, each having a different event e_i preceding the two others whose order does not matter, and therefore only three cases. Moreover, for any pair of concurrent events e **co** e', the order in which they appear in any $\mathcal{R} \in \mathcal{L}$ is irrelevant; it suffices therefore to have in $\mathcal{L}$ only one representative for any class of permutations of some set of pairwise concurrent events in $\mathcal{E}$. Therefore, the size of $\mathcal{L}$ and thus of T depends on the degree of input conflict in $\mathcal{E}$ and not on the degree of concurrency. It is known that such a performance is characteristic of methods based on partial order unfoldings.

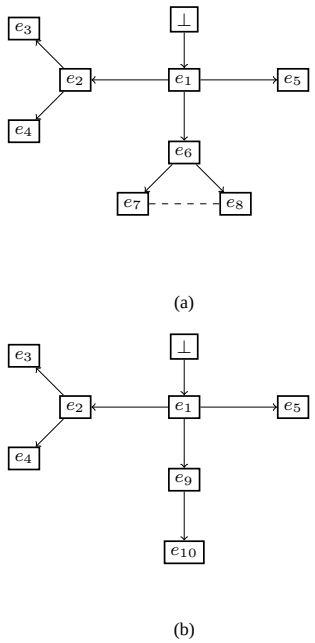


Fig. 6 Two test cases build using the IOLES in Fig. 3 and Algorithm 1.

Example 4 The test cases (a) and (b) in Fig. 6 can be obtained using Algorithm 1 and any linearizations $\mathcal{R}_1, \mathcal{R}_2$ such that $e_6 \mathcal{R}_1 e_9$ and $e_9 \mathcal{R}_2 e_6$.

Let $\mathcal{PREF}(\mathcal{E})$ be the set of all prefixes of $\mathcal{E}$, we show now that Algorithm 1 is general enough to produce a complete test suite from it.

Theorem 3 *From $\mathcal{PREF}(\mathcal{E})$ and an iics set $\mathcal{L}$ for $\mathcal{E}$, Algorithm 1 yields a complete test suite T.*

Proof Soundness: By Theorem 1 we need to prove: (1) the traces of every test case are traces of the specification; (2) every output of the specification after a trace are preserved in the test case. (1) Trace inclusion is immediate as the test case is a prefix of the unfolding of the specification. (2) For a test $\mathcal{E}_t$ and $\omega \in \text{traces}(\mathcal{E}_t)$, if an output in $\text{out}_s(\bot \textbf{ after } \omega)$ is not in $\text{out}_t(\bot \textbf{ after } \omega)$, it means either that it is in conflict with an input in $\mathcal{E}_t$, which is impossible as inputs and outputs can not be in conflict, or that its past is not already in $\mathcal{E}_t$, which is impossible since ω is a trace of $\mathcal{E}_t$.

Exhaustiveness: By Theorem 2 we need to prove that every trace is represented in at least one test case. Clearly, for all $\omega \in \text{traces}(\mathcal{E}_s)$ there exists at least one complete prefix $c \in \mathcal{PREF}(\mathcal{E})$ such that $\omega \in \text{traces}(c)$. By Proposition 1 we can find $\mathcal{R} \in \mathcal{L}$ such that this trace remains in the test case obtained by the algorithm, i.e. $\exists t \in T : \omega \in \text{traces}(t)$. □

5 Coverage Criteria for Labeled Event Structures

In the **ioco** framework and its extensions, the selection of test suites is achieved by different methods. Tests can be built in a randomized way from a canonical tester, which is a completion of the specification representing all the authorized and forbidden behaviors [23]. Closer to practice is the selection of tests according to test purposes, which represent a set of behaviors one wants to test [12]. Another method, used for symbolic transition systems for instance, is to unfold the specification until a certain testing criterion is fulfiled, and then to build a test suite covering this unfolding. Criteria for stopping the unfolding can be a given depth or state inclusion for instance [5].

5.1 Prefixes as Testing Criteria

The dynamic behavior of a Petri net is entirely captured by its unfolding, but this unfolding is usually infinite. There are several different methods of truncating an unfolding. The differences are related to the kind of information about the original unfolding one wants to preserve in the prefix.

The finite prefix algorithm depends on the notion of *cut-off* event: how long the net is unfolded. Our aim is to use

Algorithm 2 The quiescent closure of a finite prefix algorithm

Require: A 1-safe I/O Petri net $\Sigma = (T, P, F, M_0, \lambda)$ where $M_0 = \{s_1, \ldots, s_k\}$, and a *cut-off* predicate on events
Ensure: A finite prefix $\mathcal{E}^{\Theta}$ of the unfolding $\mathcal{E}$ of Σ such that $\forall \omega \in \text{traces}(\mathcal{E}^{\Theta}) : \text{out}_{\mathcal{E}^{\Theta}}(\bot \textbf{ after } \omega) = \text{out}_{\mathcal{E}}(\bot \textbf{ after } \omega)$

$\mathcal{E}^{\Theta} := (s_0, \emptyset), \ldots, (s_k, \emptyset)$
$pe := PE(\mathcal{E}^{\Theta})$
cut-off $:= \emptyset$
while $pe \neq \emptyset$ **do**
  choose an event $e = (t, B)$ in pe
  if $[e] \cap$ *cut-off* $= \emptyset$ **then**
    append to $\mathcal{E}^{\Theta}$ the event e and a condition (s, e) for every place s in $t^{\bullet}$
    $pe := PE(\mathcal{E}^{\Theta})$;
    if e is a *cut-off* event of $\mathcal{E}^{\Theta}$ **then**
      cut-off $:=$ *cut-off* $\cup \{e\}$
    end if
  else
    $pe := pe \backslash \{e\}$
  end if
end while
$pe := PE(\mathcal{E}^{\Theta})$
while $pe \cap T^{\mathcal{O}} \neq \emptyset$ **do**
  choose an event $e = (t, B)$ in $pe \cap T^{\mathcal{O}}$
  append to $\mathcal{E}^{\Theta}$ the event e and a condition (s, e) for every place s in $t^{\bullet}$
  $pe := PE(\mathcal{E}^{\Theta})$;
end while
return $\mathcal{E}^{\Theta}$

such a prefix to build test cases, therefore obtaining a finite prefix can be seen as defining a testing criterion.

As in [3], we implement a branching process of an I/O Petri net Σ as a list of nodes. A node is either a condition or an event. A condition is a pair (s, e), where s is a place of Σ and e its preset. An event is a pair (t, B), where t is a transition in Σ, and B is its preset. The possible extensions of a branching process β are the pairs (t, B) where the elements of B are pairwise in **co** relation, t is such that $\varphi(B) = {}^{\bullet}t$ and β contains no event e satisfying $\varphi(e) = t$ and ${}^{\bullet}e = B$. We denote the set of possible extensions of β by $PE(\beta)$.

As it is shown above, if the information about the produced outputs (and quiescence) is preserved in the test cases, we can prove the soundness of the test suite. Hence we aim at truncating the unfolding following a specific criterion, while preserving information about outputs and quiescence. In order to preserve this information, we follow [5] and modify the finite prefix algorithm adding all the outputs from the unfolding that the prefix enables. As there exists no cycles of outputs in the original net, this procedure terminates, yielding a finite prefix. The procedure to compute the *quiescent closure of a finite prefix* (denoted by $\mathcal{E}^{\Theta}$) is described by Algorithm 2.

The algorithm is parametric on the cutting criterion: if we change the notion of cutting event, the finite prefix obtained is different.

All-Paths-of-Length-n-Criterion. The first cut-off notion we present depends on the height of an event, defined as the length of the longest causality chain containing this event. It defines a selection criterion similar to the criterion "all paths of length n" defined in [5].

Definition 13 For a branching process *Fin*, define the *height* of an event e in *Fin* recursively by

$$\mathcal{H}(\bot) \triangleq 0$$
$$\mathcal{H}(e) \triangleq 1 + \max_{e' < e}(\mathcal{H}(e'))$$

Definition 14 Let *Fin* be a branching process. An event e is an *n-cut-off event* iff $\mathcal{H}(e) = n$.

This criterion allows us to build test cases that cover all paths of length n. However, the pertinent length n to be chosen is up to the tester.

The behavior of the system described by the specification consists usually of infinite traces. However, in practice, these long traces can be considered as a sequence of (finite) "basic" behaviors. For example, the travel agency offers few basic behaviors: (1) interaction with the server; (2) selection of insurance; and (3) selection of tickets. Any "complex" behavior of the agency is built from such basic behaviors. The longest length of these basic behaviors can be chosen as a pertinent length to unfold (see Example 6).

Inclusion Criterion. From the observation that a specification generally describes a set of basic behaviors that eventually repeat themselves, another natural criterion consists in covering the cycles of the specification. We define a criterion allowing to cover each basic behavior once, using a proper notion of *complete prefix*.

We say that a branching process β of an I/O Petri net Σ is *complete* if for every reachable marking M there exists a configuration C in β such that:

1. $Mark(C) = M$ (i.e. M is represented in β), and
2. for every transition t enabled by M there exists $C \cup \{e\} \in \mathcal{C}(\beta)$ such that e is labeled by t.

The following notion corresponds to the inclusion criterion where each cycle in unfolded once.

Definition 15 Let *Fin* be a branching process. An event e is an *inclusion cut-off event* iff *Fin* contains an event $e' \leq e$ such that *Mark*$([e']) =$ *Mark*$([e])$.

Note that the completeness of a prefix does not imply that the information about outputs and quiescence is preserved, so Algorithm 2 still is necessary to build its quiescent closure.

Example 5 Consider Fig. 7, we have that *Fin* is complete, but the expected outputs are not part of the prefix. We expect that o_1 is produced by the system after i_2 and i_4, i.e. $\text{out}_{\mathcal{E}}(\bot \textbf{ after } (i_2 \cdot i_4)) = \{o_1\}$, but this is not the case in *Fin*, i.e. $o_2 \notin \text{out}_{Fin}(\bot \textbf{ after } (i_2 \cdot i_4)) = \{\delta\}$.

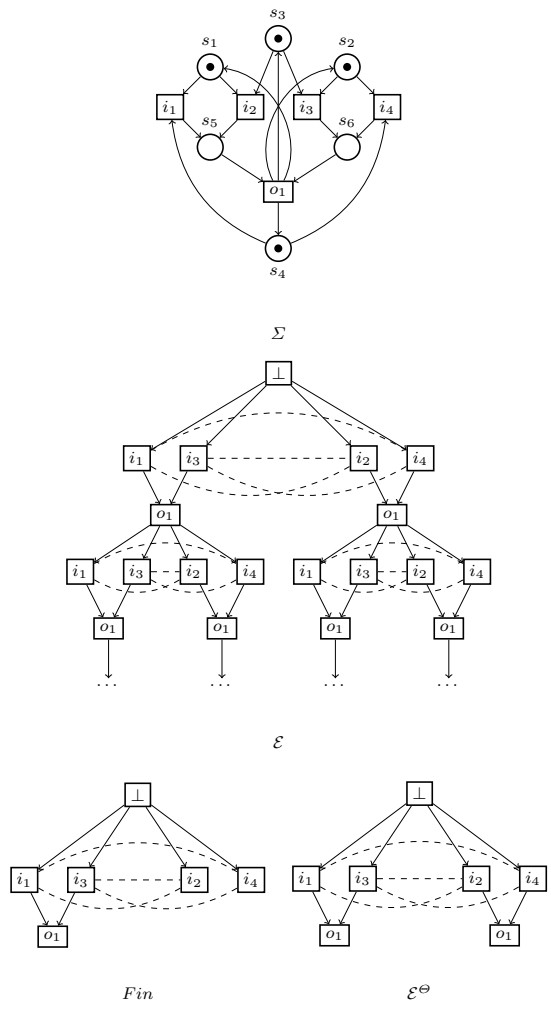


Fig. 7 I/O Petri net Σ, part of its unfolding $\mathcal{E}$, a complete finite prefix *Fin* and its quiescent closure $\mathcal{E}^{\Theta}$.

It has been proved that the prefix obtained in [3] is complete, therefore such result also holds for the prefix obtained by our algorithm when we consider the inclusion criterion. However the notion of n-cut-off event needed for the "all paths of length n" criterion does not guarantee completeness (it may be the case that not every marking is represented in the prefix).

k-inclusion Criterion. A natural extension of the previous criterion consists in unfolding each cycle several times. We present below the k-inclusion criterion which together with Algorithm 2 leads to a complete prefix unfolding each cycle k times and preserving outputs and quiescence.

Definition 16 Let *Fin* be a branching process. An event e is a *k-inclusion cut-off event* iff *Fin* contains a family of k events $\{e_i\}_{i \leq k}$ such that $e_i \leq e$ and $\mathit{Mark}([e_i]) = \mathit{Mark}([e])$.

Obviously a 1-inclusion cut-off event is an inclusion cut-off event in the sense of Definition 15.

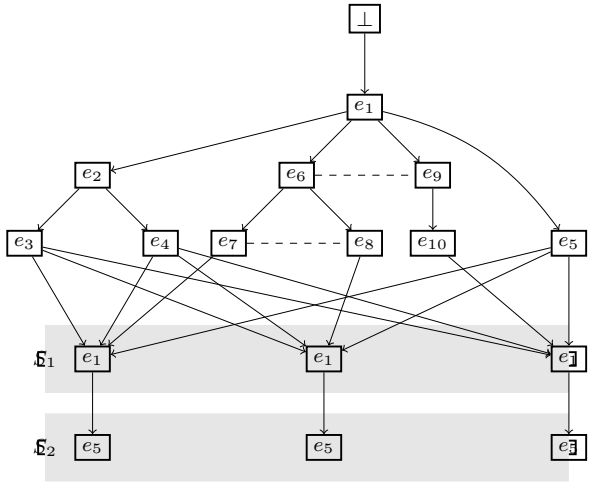


Fig. 8 *All-paths-of-length-n* criterion.

Example 6 (Determining length n to fulfill the inclusion criterion)

Consider the unfolding of the travel agency in Fig. 8. The nodes in grey in set S_1 are those marked as 1-inclusion cut-off, meaning that the prefix ending with these nodes fulfills the 1-inclusion criterion. As all these events have the same height 4, this prefix also fulfills the "all paths of length 4" criterion. Finally we obtain the quiescent closure adding nodes of set S_2 in order to preserve outputs.

The following result is central and will help proving soundness of the test suites proposed below.

Theorem 4 *Let $\mathcal{E} \in \mathcal{IOLES}(L)$ and $\mathcal{E}^{\Theta}$ the quiescent closure of its finite prefix obtained either by the k-inclusion or the "all paths of length n" criterion. Then*

1. $\mathit{traces}(\mathcal{E}^{\Theta}) \subseteq \mathit{traces}(\mathcal{E})$
2. $\forall \omega \in \mathit{traces}(\mathcal{E}^{\Theta}) : \mathit{out}_{\mathcal{E}^{\Theta}}(\perp \textbf{\textit{after}}\ \omega) = \mathit{out}_{\mathcal{E}}(\perp \textbf{\textit{after}}\ \omega)$

Proof 1) is immediate since $\mathcal{E}^{\Theta}$ is a prefix of $\mathcal{E}$. Since only the outputs produced after the traces of $\mathcal{E}^{\Theta}$ are considered, 2) follows by its construction. □

The test suites constructed based on the k-inclusion or the "all paths of length n" criterion are sound:

Theorem 5 *Let Σ_s be the specification of a system and $\mathcal{E}_s$ the IOLES of its unfolding. Any test suite constructed using Algorithm 1 and $\mathcal{E}_s^{\Theta}$ as an input is sound for $\mathcal{E}_s$ w.r.t* ***co-ioco****.*

Proof By Theorem 1 we need to prove that any trace of a test case $\mathcal{E}_t$ is a trace of $\mathcal{E}_s$ (which is trivial as $\mathcal{E}_t$ is a prefix of $\mathcal{E}_s^{\Theta}$ and therefore of $\mathcal{E}_s$) and that outputs and quiescence produced after any trace ω of such a test are preserved. The events of $\mathcal{E}_s^{\Theta}$ that are added to $\mathcal{E}_t$ are all the events whose past is already in $\mathcal{E}_t$ and which are not in immediate conflict with an input. An output cannot be in immediate conflict with an input by Assumption 2, so all the outputs whose

past is already in $\mathcal{E}_t$ are added. So all the outputs from $\mathcal{E}_s^{\Theta}$ after a trace ω are preserved and by Theorem 4 we have $\forall \omega \in \text{traces}(\mathcal{E}_t) : \text{out}_t(\bot \textbf{ after } \omega) = \text{out}_s(\bot \textbf{ after } \omega)$. □

Example 7 The IOLES of Fig. 3 is a complete prefix of the unfolding of the net in Fig. 2 and can be obtained using Algorithm 2. We saw in Example 4 how to build test cases that cover such a complete prefix. Thus the test cases of Fig. 6 form a sound test suite that covers the specification according to our inclusion criterion.

Remark 3 Let us note that we lose completeness of the test suite we build by selecting test cases from the original complete test suite. While in a model-checking context, finite prefixes have been shown to contain enough information to verify global properties [2], in a testing context, it is impossible for a finite test suite to be complete with respect to a specification containing infinite behaviors. Since the implementation is not known, we do not have any information about its space state, therefore we can only *assume* that any incorrect behavior will occur in a finite path of the system.

5.2 Comparing Different Criteria

The notion of cut-off presented in [3] is more general that ours and depends on the notion of an *adequate order*. An adequate order $\prec$ is a well-founded partial order on the finite configurations of the unfolding that refines set inclusion and is preserved by extensions, i.e. if $C_1 \prec C_2$ and $\mathit{Mark}(C_1) = \mathit{Mark}(C_2)$ then $C_1 \cup E \prec C_2 \cup E$ for all finite extensions E. An event e is marked as a cut-off if there exists a configuration C in the prefix such that $C \prec [e]$ and $\mathit{Mark}(C) = \mathit{Mark}([e])$. Adequate orders lead usually to smaller prefixes which is something desirable in model checking, but this not always the case for testing.

Example 8 Consider Fig. 9: (a) is a net; (b) the prefix obtained with an adequate order $\prec_r$ presented in [3] and adding the corresponding outputs; (c) the prefix obtained by our 1-inclusion criterion. Prefix (b) is smaller than (c), however it is of less quality w.r.t. its ability to detect bugs. Consider a non conformant implementation that accepts $?b_1$ followed by $!d_1$ and then deadlocks, i.e. $\text{poss}_i(\bot \textbf{ after } ?b_1!d_1) = \{\}$ while $\text{poss}_s(\bot \textbf{ after } ?b_1!d_1) = \{?a_2, ?b_2\}$. The test suite that we obtain from (b) is $T_1 = \{?b_1!c_1, ?b_1!d_1, ?a_1!d_1, ?a_1!c_1?b_2, ?a_1!c_1?a_2\}$ which passes the implementation, i.e. non conformance is not detected. Let T_2 be the test suite obtained from (c) and Algorithm 1, then $?b_1!d_1?a_2 \in T_2$ and we have a test case that makes the test execution fail, i.e. non conformance is detected.

We need therefore a way to compare the testing power of different prefixes. We can follow the k-inclusion criterion and consider the number of times a marking is present as a measure for the quality of the test suite, however this does not consider the "cost" of producing such test suite: as it is shown in Fig. 9, our prefixes can be exponentially bigger than those constructed using adequate orders. We need therefore to balance between the testing power and the size of the prefix. As it is shown in Fig. 10, the size in which each branch increases for obtaining a certain k-inclusion is not always the same. If we want to increase k-inclusion by adding $!r_1$, we need to make new copies of p_2 and then only two new events (those corresponding to $?a_1$ and $?b_1$) need to be added. However if we also want to increase the inclusion by adding $!r_2$, 14 events (those presented in Fig. 9(c)) need to be added. This kind of explosion is one of the disadvantages of unfoldings, however there exist other ways to unfold the net to avoid such kind of explosion (see for example [14]).

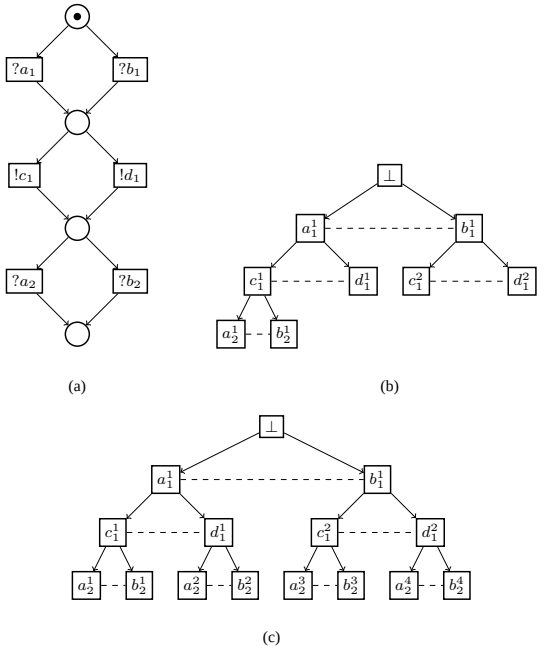


Fig. 9 Complete prefixes with adequate orders.

We define the coverage of a configuration to be measured by the number of times its corresponding marking is represented in any proper subset:

$$\mathcal{C}\mathit{ov}(C) \triangleq |\{C' \subset C \mid \mathit{Mark}(C) = \mathit{Mark}(C')\}|$$

In order to take into account the "cost" of building a larger prefix to increase the coverage of a configuration, we define the quality of a configuration C in a given prefix *Fin* to be its coverage divided by the number of events of the prefix:

$$\mathcal{Q}(C) \triangleq \frac{\mathcal{C}\mathit{ov}(C)}{|\mathit{Fin}|}$$

Finally, the quality of a prefix will be the smallest quality of its maximal configurations.

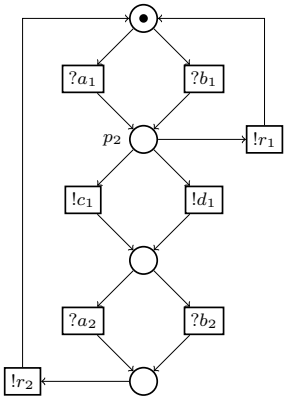


Fig. 10 Exponential grow of unfoldings.

Definition 17 Let *Fin* be a finite prefix of the unfolding, we define the quality of *Fin* as

$$\mathcal{Q}(\mathit{Fin}) \triangleq \min_{C \in \Omega(\mathit{Fin})} \mathcal{Q}(C)$$

In Fig. 10 we can see that the quality of configurations where $!r_2$ is unfolded are smaller than those that unfold $!r_1$ (as they contain more events). Therefore the quality of the finite prefixes (and that of the test cases obtained from it) does not depend on how many times we unfold transition $!r_1$, but on the number of times we unfold $!r_2$.

6 Conclusion and Future Work

We have presented a testing framework and a test generation algorithm for true concurrency specifications of distributed and concurrent systems. Our test selection criteria are based on the quiescent closure of finite prefixes of the unfolding of the specification; they allow to select, among all possible test cases, those covering all paths of length n or those traversing each basic behavior a certain number of times.

Let us point out that the testing approach we followed in this article is mostly theoretical, since concurrency is not easily observable at a global point of view. We defined the notions of test case and execution of a test case from a global point of control and observation, where concurrency is always kept explicit. However, in practice, such global test cases are not meant to be actually executed globally. They would rather be projected onto the different processes to be executed locally, in order to make the observation of concurrency possible. Our approach here is to study the testing problem from a centralized point of view, as a basis to the distributed testing problem: the global conformance relation we defined is the relation we want to still be able to test in a distributed way (with local control and observation), and the global test cases are the basis for the construction of distributed tests.

In practice, the global control and observation assumption may not be satisfied and we can only observe the system partially, i.e. only the behavior of a local process is observed. In a distributed testing environment, a local tester interacts with each process. If we are in a pure distributed testing setting, there is no global clock. We are currently studying under which assumptions global conformance can be decided by the conformance of every single process. Another possibility is to weaken the conformance relation to consider a distributed architecture as it is in the case of the **dioco** framework of Hierons et al. [10] for multi-port IOTS.

Future technical studies include the question whether it is possible to drop assumptions 1 and 2 under a bounded fairness assumption, meaning that in a given configuration, all the different events will eventually occur if the experiment is repeated a bounded number of times. However under such an assumption, controllability of test cases must be ensured during their construction.

References

1. Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *IEEE Computer Security Foundations Symposium*, pages 107–121. IEEE Computer Society, 2010.
2. Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5), 2006.
3. Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. In *Tools and Algorithms for Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 1996.
4. Alain Faivre, Christophe Gaston, Pascale Le Gall, and Assia Touil. Test purpose concretization through symbolic action refinement. In *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2008.
5. Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *Testing of Software and Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.
6. Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing input/output partial order automata. In *Testing of Software and Communicating Systems*, volume 4581 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2007.
7. Lex Heerink and Jan Tretmans. Refusal testing for classes of transition systems with inputs and outputs. In *Formal Description Techniques for Distributed Systems and Communication Protocols*, volume 107 of *IFIP Conference Proceedings*, pages 23–38. Chapman & Hall, 1997.
8. Matthew Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.
9. Olaf Henniger. On test case generation from asynchronously communicating state machines. In *Testing of Communicating Systems*, IFIP, pages 255–271. Springer, 1997.
10. Robert M. Hierons, Mercedes G. Merayo, and Manuel Núñez. Implementation relations for the distributed test architecture. In *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2008.
11. Claude Jard. Synthesis of distributed testers from true-concurrency models of reactive systems. *Information & Software Technology*, 45(12):805–814, 2003.

12. Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7:297–315, 2005.
13. Thierry Jéron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240:167–184, 2009.
14. Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. Merged processes: a new condensed representation of Petri net behaviour. *Acta Informatica*, 43(5):307–330, 2006.
15. Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.
16. Grégory Lestiennes and Marie-Claude Gaudel. Test de systèmes réactifs non réceptifs. *Journal Européen des Systèmes Automatisés*, 39(1-2-3):255–270, 2005.
17. Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.
18. Jan Peleska and Michael Siegel. From testing theory to test driver implementation. In *Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 538–556. Springer, 1996.
19. Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Conformance relations for labeled event structures. In *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.
20. Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Unfolding-based test selection for concurrent conformance. In *International Conference on Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2013.
21. Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.
22. Roberto Segala. Quiescence, fairness, testing, and the notion of implementation. *Information and Computation*, 138(2):194–210, 1997.
23. Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.
24. Andreas Ulrich and Hartmut König. Specification-based testing of concurrent systems. In *Formal Description Techniques for Distributed Systems and Communication Protocols*, volume 107 of *IFIP Conference Proceedings*, pages 7–22. Chapman & Hall, 1998.
25. Gregor von Bochmann, Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing systems specified as partial order input/output automata. In *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2008.
26. Glynn Winskel. Petri nets, morphisms and compositionality. In *Applications and Theory in Petri Nets*, pages 453–477, 1985.