Model Based Testing for Concurrent Systems with Labeled Event Structures

Hernán Ponce de León, Stefan Haar, Delphine Longuet

Model Based Testing for Concurrent Systems with Labeled Event Structures

Hernán Ponce de León[1], Stefan Haar[1], and Delphine Longuet[2]

[1] INRIA and LSV, École Normale Supérieure de Cachan and CNRS, France
ponce@lsv.ens-cachan.fr
stefan.haar@inria.fr
[2] Univ Paris-Sud, LRI UMR8623, Orsay, F-91405
longuet@lri.fr

Abstract. We propose a theoretical testing framework and a test generation algorithm for concurrent systems that are specified with true concurrency models, such as Petri nets or networks of automata. The semantic model of computation of such formalisms are labeled event structures, which allow to represent concurrency explicitly. The activity of testing relies on the definition of a conformance relation that depends on the observable behaviors on the system under test. The **ioco** type conformance relations for sequential systems rely on the observation of sequences of inputs and outputs and blockings. However these relations are not capable of capturing and exploiting concurrency of non sequential behavior. We propose an extension of the **ioco** conformance relation for labeled event structures, named **co-ioco**, which allows to deal with explicit concurrency. We give an algorithm to build test cases from a given specification and prove that the generated test suite is complete for **co-ioco**.

1 Introduction

Model-based testing. One of the most popular formalisms studied in conformance testing is that of *labeled transition systems* (LTS). A labeled transition system is a structure consisting of states and transitions labeled with actions from one state to another. This formalism is usually used for modeling the behavior of sequential processes and as a semantical model for various formal languages such as CCS [1], CSP [2], SDL [3] and LOTOS [4].

Several testing theories have been defined for labeled transition systems [5–11]. A formal testing framework relies on the definition of a conformance relation which formalizes the relation that the system under test and its specification must verify. Depending on the nature of the possible observations of the system under test, several conformance relations have been defined for labeled transition systems. The relation of *trace preorder* (trace inclusion) is based on the observation of possible sequences of actions only. It was refined into the *testing preorder*, that requires not only the inclusion of the implementation traces in those of the specification, but also that any action refused by the implementation should be refused by the specification [5, 12]. A practical modification of the

testing preorder was presented in [7], which proposed to base the observations on the traces of the specification only, leading to a weaker relation called **conf**. A further refinement concerns the inclusion of quiescent traces as a conformance relation (e.g. Segala [10]). Moreover, Tretmans [11] proposed the **ioco** relation, which refines **conf** with the observation of blockings (quiescence).

The **ioco** conformance relation is defined for input-output labeled transition systems which are LTS where stimuli received from the environment (inputs) are distiguished from answers given by the system (outputs). It relies on two kinds of observation: traces, that are sequences of inputs and outputs, and quiescence, which is the observation of a blocking of the system (the system will not produce outputs anymore or is waiting for an input from the environment to produce some). A system under test conforms to its specification with respect to **ioco** if after any trace of the specification that can be executed on the system, the observable outputs and blockings of the system are possible outputs and blockings in the specification.

The testing theory based on the **ioco** conformance relation has now become a standard and is used as a basis in several testing theories for extended state-based models. Let us mention here the works on restrictive transition systems [13, 14], symbolic transition systems [15, 16], timed automata [17], and multi-port finite state machines [18].

Model-based testing of concurrent systems. However, the testing frameworks based on labeled transitions do not very well support the testing from concurrent specifications, in which some pairs of events can be specified to occur in arbitrary order, or jointly. At the same time, it is obvious — and generally acknowledged — that *concurrent* models are often advantageous and natural, and sometimes inevitable.

The design of complex systems is often modular and proceeds by first developing local, automata-type components, and then composing these building blocks by keeping all elements that are local to one component as such; only the synchronization on shared actions is needed, performing local state changes on two or more components at once. This leads naturally to modeling the composite system as a *network of finite automata*, a formal class of models that can be captured equivalently by *safe Petri nets.*

More generally, concurrency can arise in the design of a for different reasons. First, two events may be physically localized on different components, and thus be "naturally" independent of one another; this distribution is then part of the system construction, and may be either part of the specification or an implementation choice. Second, the specification may not care about the order in which two actions are performed *on the same component*, 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 the distribution of components. Therefore, in general, no order

can be observed between events occuring on different components. We will see how this affects the testing of such systems.

In order to test concurrent systems, the exhaustive testing of all interleavings for a set of concurrent transitions is prohibitively slow; at the same time, the problem of state space explosion can be addressed efficiently by using models involving *local* states and *local* actions only, leaving the order of occurrence unspecified for events that occur independently of one another in parallel on different components of the systems. That is, instead of storing and exploring the entire set of interleaved behaviors, one can more efficiently use partially ordered structures.

Let us add that true concurrency models are not only promising for practical reasons, but also are more adequate in reflecting the actual structure of distributed systems; they also tend to be more intuitive and accessible for both designers and implementers, in particular if modularity as above can be exploited.

While the passage to concurrent models has been successfully performed in other fields of formal analysis such as model checking or diagnosis, *testing* has embraced concurrent models somewhat more recently.

However, already in [19], Ulrich and König propose a framework for testing concurrent systems specified by communicating labeled transition systems. They define a concurrency model called behavior machines that is an interleaved-free and finite description of concurrent and recursive behavior, which is a sound model of the original specification. Their testing framework relies on a conformance relation defined by labeled partial order equivalence, and allows to design tests for each component from a labeled partial order representing an execution of the behavior machine. In another direction, Haar et al [20, 21] generalized the basic notions and techniques of I/O-sequence based conformance testing on a generalized I/O-automaton model where partially ordered patterns of input/output events were admitted as transition labels. An important practical benefit of true-concurrency models here is an overall complexity reduction, despite the fact that checking partial orders requires in general multiple passes through the same labelled transition, so as to check for presence/absence of specified order relations between input and output events. In fact, if the system has n parallel and interacting processes, the length of checking sequences increases by a factor that is polynomial in n. At the same time, the overall size of the automaton model (in terms of the number of its states and transitions) shrinks exponentially if the concurrency between the processes is explicitly modeled. This feature indicates that with increasing size and distribution of SUTs in practice, it is computationally wise to seek alternatives for the direct sequential modeling approach.

However, the models of [20, 21] still force us to maintain a sequential automaton as the system's skeleton, and to include synchronization constraints (typically: that all events specified in the pattern of a transition must be completed before any other transition can start), which limit both the application domain and the benefits from concurrency modeling. The approach that we fol-

low here, continuing the work started in [22], proposes a formal framework for testing concurrent systems from true-concurrency models in which no synchronization on global states is required.

Our framework and contributions. We use a canonical semantic model for concurrent behavior, *labeled event structures*, providing a unifying semantic framework for system models such as Petri nets, networks of automata, communicating automata, or process algebras; we abstract away from the particularities of system specification models, to focus entirely on behavioral relations.

The underlying mathematical structure for the system semantics is given by *event structures* in the sense of Winskel et al [23]. Mathematically speaking, they are particular partially ordered sets, in which order between two events e and e' indicates precedence, and where any two events e and e' that are *not* ordered may be either

- in *conflict*, meaning that in any evolution of the system in which e occurs, e' *cannot* occur; or
- *concurrent*, in which case they may occur in the same system run, without a temporal ordering, i.e. e may occur before e', after e', or simultaneously.

Event structures arise naturally under the partial order unfolding semantics for Petri nets [23], and also as a natural semantics for process algebras (see e.g. [24]). The state reached after some execution is represented by a *configuration* of the event structure, that is a conflict-free, history-closed set. The use of partial order semantics provides richer information and finer system comparisons than the interleaved view.

In [22], we proposed an extension of the **ioco** conformance relation to labeled event structures, named **co-ioco**, which takes concurrency explicitly into account. In particular, it forces events that are specified as concurrent to remain concurrent in the implementation.

In this paper, we refine this conformance relation in several ways. First we take into account silent actions in the specification, that are internal actions of the system (opposed to interactions with the environment that are inputs and outputs). More importantly, since we cannot observe concurrency between outputs (or in other words, we cannot ensure that two outputs observed after the same input do not depend on each other), we refine the **co-ioco** conformance relation in order to allow outputs specified as concurrent to be implemented sequentially. Another refinement of the conformance relation is the observation of possible and impossible inputs of the system under test. We want the system to implement at least the possible inputs of the specification, or in other words, if an input is not possible in the system under test, it should not be possible in the specification either. These three main refinements lead to a new definition of the **co-ioco** conformance relation.

The main contribution of this paper is the definition of a whole framework for testing concurrent systems from labeled event structures. Besides the **co-ioco** conformance relation, we define 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. This framework is presented according to the following structure.

Structure of the paper. In the next section, we will introduce several basic notions such as input output labeled event structures *(IOLES)* and a novel partial order semantics for IOLES, that allows to "relax" concurrency. In section 3, we develop the *observational* framework for IOLES, introducing in particular the notions of quiescence and refusals for partial order semantics. Section 4 is dedicated to the definition, discussion and characterization of the input-output conformance relation **co-ioco** , refining the **co-ioco** relation from our previous paper [22]. Section 5 develops the definitions of test cases and test suites, characterizing completeness, soundness and exhaustiveness and proposes an algorithm that builds a complete test suite, thus completing the contributions of the paper before Section 6 concludes.

2 Input/Output Labeled Event Structures

2.1 Syntax

We shall be using event structures following Winskel et al [23] to describe the dynamic behavior of a concurrent system. In this paper we will consider only prime event structures [25], a subset of the original model which is sufficient to describe concurrent models (therefore we will simply call them event structures), and we label their events with actions over a fixed alphabet L. As it is usual with reactive systems, we want to distinguish between the controllable actions (inputs proposed by the environment) and the observable ones (outputs produced by the system), leading to *input-output labeled event structures.*

Definition 1 (Input/Output Labeled Event Structure). *An input/output labeled event structure (IOLES) over an alphabet $L = L_{\mathcal{I}} \uplus L_{\mathcal{O}}$ is a 4-tuple $\mathcal{E} = (E, \leq, \#, \lambda)$ where*

- *E is a set of events,*
- *$\leq \subseteq E \times E$ is a partial order (called* causality*) satisfying the property of* finite causes*, i.e. $\forall e \in E : |\{e' \in E \mid e' \leq e\}| < \infty$,*
- *$\# \subseteq E \times E$ is an irreflexive symmetric relation (called* conflict*) satisfying the property of* conflict heredity*, i.e. $\forall e, e', e'' \in E : e \# e' \wedge e' \leq e'' \Rightarrow e \# e''$,*
- *$\lambda : E \to L \cup \{\tau\}$ is a labeling mapping.*

We denote the class of all IOLES over L by $\mathcal{IOLES}(L)$.

Given an event e, its past is defined as $[e] \triangleq \{e' \in E \mid e' \leq e\}$. Two given events $e, e' \in E$ are said to be *concurrent*, written e **co** e', iff neither $e \leq e'$ nor $e' \leq e$ nor $e \# e'$ hold. The kind of concurrent systems that we consider are distributed. The architecture of such systems allows to distinguish different components and there is a way to distinguish which component each concurrent action belongs to.

Assumption 1 *We will only consider systems where concurrent events are labeled by different actions, i.e.* $\forall e, e' \in E : e \ \mathbf{co}\ e' \Rightarrow \lambda(e) \neq \lambda(e')$.

The special label $\tau \notin L$ represents an unobservable (also said internal or silent) action. The sets of input and output events are defined by $E^{\mathcal{I}} \triangleq \{e \in E \mid \lambda(e) \in L_{\mathcal{I}}\}$ and $E^{\mathcal{O}} \triangleq \{e \in E \mid \lambda(e) \in L_{\mathcal{O}}\}$. When it is clear from the context, we will refer to an event by its label.

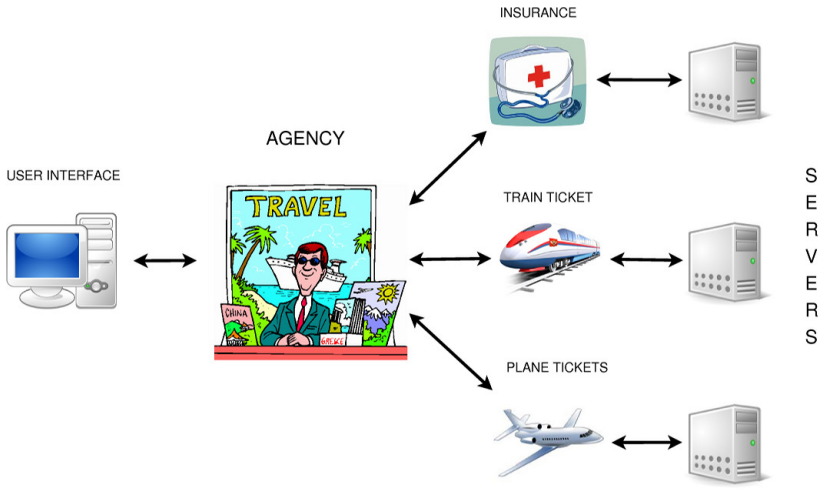


Fig. 1. Informal specification of a travel agency.

Example 1. The behavior of a travel agency is informally presented in Figure 1 and can be formally specified by the IOLES s presented in Figure 2, where causality and conflict are represented by $\rightarrow$ and - - - respectively, ? 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 (?*insurance*) and a train ticket (?*train*) or a plane ticket (?*plane*). If a plane ticket was chosen, its price is sent to the user (!*pl_price*). If a train ticket was selected, the agency can internally decide (τ) what price to propose: a first class (!*tr_price* 1) or a second class (!*tr_price* 2) one. The insurance choice is followed by its price (!*ins_price*) and some extra data that is sent to the user (!*ins_data*).

The data can not 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** ?*insurance*), but only one ticket can be chosen (?*train* # ?*plane*). From the conflict heredity property, we have that only one ticket's price can be produced (!*tr_price* 1 # !*pl_price* and !*tr_price* 2 # !*pl_price*).

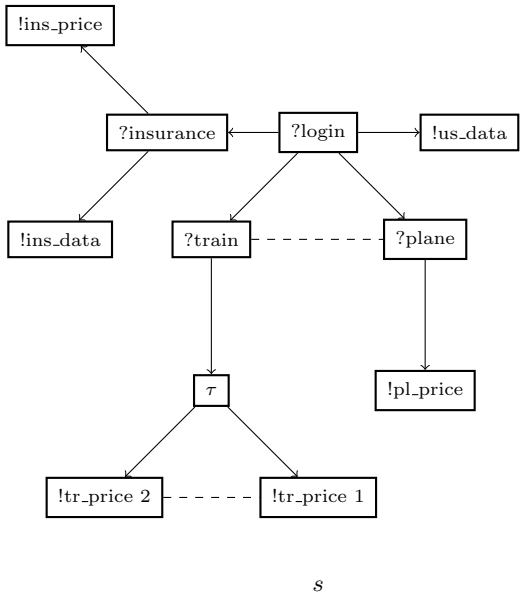


Fig. 2. Input/output labeled event structure of a travel agency.

Most of the specification languages allow some way of modeling choice. As conflict is inherited w.r.t to causal dependency, it can not always be interpreted as a choice for the system, as it is illustrated by the example below.

Example 2. In Figure 2 we can see that !*tr_price* 1 and !*pl_price* are in conflict (by hierarchy), but any computation that can continue directly by !*tr_price* 1, can not continue by !*pl_price* as the conflict was solved by the choice of ?*train* instead of ?*plane* and then !*pl_price* is not possible.

The notion of choice is captured by the *immediate conflict* relation.

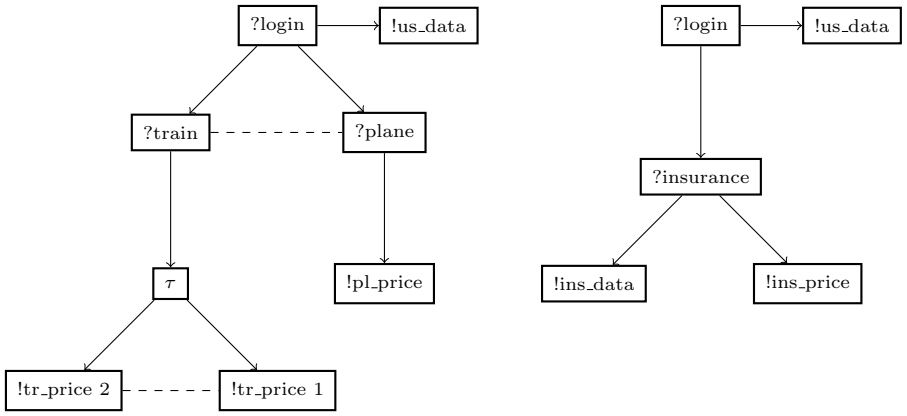
Definition 2 (Immediate Conflict). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$ *and* $e_1, e_2 \in E$*. Events* e_1 *and* e_2 *are said in immediate conflict, written* $e_1 \overline{\#} e_2$*, iff*

$$[e_1] \times [e_2] \cap \# = \{(e_1, e_2)\}$$

The behavior of a subsystem is given by a *prefix* of the original system. As causality represents the events that should occur before a given event e, the past of e should be part of the prefix.

Definition 3 (Prefix). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$*. A prefix of* $\mathcal{E}$ *is an IOLES* $\mathcal{E}' = (E', \leq', \#', \lambda')$ *where*

- $E' \subseteq E$ *such that* $\forall e \in E' : [e] \subseteq E'$,
- $\leq' = \leq \cap (E' \times E')$,



s_2 s_3

Fig. 3. Two prefixes of the travel agency.

- $\#' = \# \cap (E' \times E')$, *and*
- $\lambda' = \lambda_{|E'}$

Example 3. In Figure 3, s_2 specifies the behavior of a travel agency that sells tickets, but not insurances, while the agency in s_3 only sells insurances. We can see that s_2 and s_3 are prefixes of s.

2.2 Semantics

In order to give a semantics to event structures, we have to define the notion of execution, which relies on the notion of state, named configuration in labeled event structures.

A computation state of an event structure is called a *configuration* and is represented by the set of events that have occurred in the computation. If an event is present in a configuration, then so are all the events on which this event causally depends (causal closure). Moreover, a configuration obviously does not contain conflicting events (conflict freedom). This is captured by the following standard definition [25].

Definition 4 (Configuration). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$. A configuration of $\mathcal{E}$ is a set of events $C \subseteq E$ where*

- *C is causally closed: $e \in C \Rightarrow \forall e' \leq e : e' \in C$, and*
- *C is conflict-free: $\forall e, e' \in C : \neg(e \# e')$.*

A configuration is the equivalence class of all the possible interleavings. The initial configuration of $\mathcal{E}$, denoted by $\bot_{\mathcal{E}}$, is the empty set of events. We denote the set of all the configurations of $\mathcal{E}$ by $\mathcal{C}(\mathcal{E})$.

Example 4. As explained above, after the user has logged in, the selections can be made while his information is sent to the server, i.e. $\{?login, !us\_data, ?insurance, ?train\} \in \mathcal{C}(s)$, but a train and a plane ticket can not be selected in the same execution, i.e. $?plane, ?train \in C \Rightarrow C \notin \mathcal{C}(s)$. The configuration $\{?login, !us\_data, ?insurance, !ins\_price, !ins\_data, ?train, \tau, !tr\_price\ 1\}$ is maximal (w.r.t $\subseteq$) as the remaining events are in conflict with the ones in the configuration.

Note that the testing activity depends on the interaction between the tester and the system and that such interaction becomes impossible if we allow the system to have infinitely many occurrences of silent or output actions without input ones.

Assumption 2 *We will only consider systems that can not diverge by infinite occurrences of silent or output actions, i.e. any configuration having infinitely many silent or output actions should have infinitely many input ones.*

This assumption is classical in model-based testing frameworks, it is necessary to be able to identify the blockings of the system under test.

The definition of the notion of execution for an event structure is not straightforward since it relies on the chosen semantics for concurrency [26]. The presence of explicit concurrency in a specification may be interpreted in several ways. In an early stage of specification, concurrency between events may be used as underspecification, leaving the choice of the actual order between events for further refinements. The concurrency between events can also be interpreted as if any order of such events is considered correct (interleaving semantics). In a distributed system however, concurrent events in the specification may be meant to remain concurrent in the implementation, because they are destined to occur in different components executed in parallel (partial order semantics).

We are interested in testing distributed systems where concurrent actions occur in different components of the system. For this reason, we want to keep concurrency explicit, meaning that we do not want to impose an order of execution between concurrent events. *Labeled partial orders* can then be used to represent executions of such systems.

Definition 5 (Labeled partial order). *A labeled partial order over an alphabet L is a tuple $lpo = (E, \leq, \lambda)$, where*

- *E is a set of events,*
- *$\leq$ is a reflexive, antisymmetric, and transitive relation, and*
- *$\lambda : E \to L \cup \{\tau\}$ is a labeling mapping.*

We denote the class of all labeled partial orders over L by $\mathcal{LPO}(L)$.

As we can only observe the ordering between the labels and not between the events, we should consider partial orders respecting such an order as equivalent. Two labeled partial orders are *isomorphic* iff there exists a bijective function that preserves ordering and labeling.

Definition 6 (Isomorphic LPOs). *Let* $lpo_1 = (E_1, \leq_1, \lambda_1), lpo_2 = (E_2, \leq_2, \lambda_2) \in \mathcal{LPO}(L)$. *A bijective function* $f : E_1 \to E_2$ *is an* isomorphism *between* lpo_1 *and* lpo_2 *iff*

- $\forall e, e' \in E_1 : e \leq_1 e' \Leftrightarrow f(e) \leq_2 f(e')$
- $\forall e \in E_1 : \lambda_1(e) = \lambda_2(f(e))$

Two labeled partial orders lpo_1 *and* lpo_2 *are isomorphic if there exists an isomorphism between them.*

Definition 7 (Partially ordered multisets). *A* partially ordered multiset *(pomset) is the isomorphic class of some LPO. We will represent such class by one of its objects. We denote the class of all pomsets by* $\mathcal{POMSET}(L)$.

Single actions can be represented by trivial pomsets. When it is clear from the context, we will use $\cdot$ to express causality and **co** to represent the pomset whose elements are unordered. The pomset μ_4 of Figure 4 can be represented by $?login \cdot !us\_data \cdot ?insurance \cdot (!ins\_price$ **co** $!ins\_data)$.

However, as output actions are uncontrollable in a testing environment, we can not impose the concurrency of such actions to remain in the implementation, and then μ_1 and μ_2 or μ_3, μ_4 and μ_5 should be treated as equal. For such a reason we will require in the implementation relation that the partial order semantics be respected, up to adding order between concurrent outputs, or allowing the occurrence of an output to precede the occurrence of a concurrent input.

Definition 8 (Implementation relation with Relaxing concurrency). *Let* $\mu_1, \mu_2 \in \mathcal{POMSET}(L)$, *we have that* $\mu_1 \sqsubseteq \mu_2$ *iff there exist* $lpo_1 = (E, \leq_{\mu_1}, \lambda) \in \mu_1$ *and* $lpo_2 = (E, \leq_{\mu_2}, \lambda) \in \mu_2$ *such that*

- $\leq_{\mu_1} \cap (E^{\mathcal{I}} \times E) \;=\; \leq_{\mu_2} \cap (E^{\mathcal{I}} \times E)$
- $\leq_{\mu_1} \cap (E^{\mathcal{O}} \times E) \;\subseteq\; \leq_{\mu_2} \cap (E^{\mathcal{O}} \times E)$

Example 5. We see in Figure 4 that μ_2 adds some ordering between an output action and an input one, i.e. $?login \leq !us\_data$, from μ_1, therefore $\mu_1 \sqsubseteq \mu_2$. The same order is added from μ_3 in μ_4 and some ordering between output actions is added is μ_5. Finally $\mu_3 \sqsubseteq \mu_4$ and $\mu_4 \sqsubseteq \mu_5$.

As explained above, an execution of an event structure can be represented by a pomset, where the same pomset can reflect different executions in which concurrency can be relaxed, leading to the following notion of *relaxed executions.*

Definition 9 (Relaxed execution). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L), \mu, \mu'' \in \mathcal{POMSET}(L)$ *and* $C, C' \in \mathcal{C}(\mathcal{E})$, *we define*

$$C \xrightarrow{\mu} C' \quad \triangleq \exists \mu' \sqsubseteq \mu, lpo = (E_{\mu'}, \leq_{\mu'}, \lambda_{\mu'}) \in \mu', A \subseteq E \backslash C : C' = C \cup A, A = E_{\mu'}, \leq \cap (A \times A) = \leq_{\mu'} \text{ and } \lambda_{|A} = \lambda_{\mu'}$$

$$C \xrightarrow{\mu \cdot \mu''} C'' \triangleq C \xrightarrow{\mu} C' \text{ and } C' \xrightarrow{\mu''} C''$$

$$C \xrightarrow{\mu} \quad \triangleq \exists C' : C \xrightarrow{\mu} C'$$

We say that μ *is a relaxed execution of* C *if* $C \xrightarrow{\mu}$.

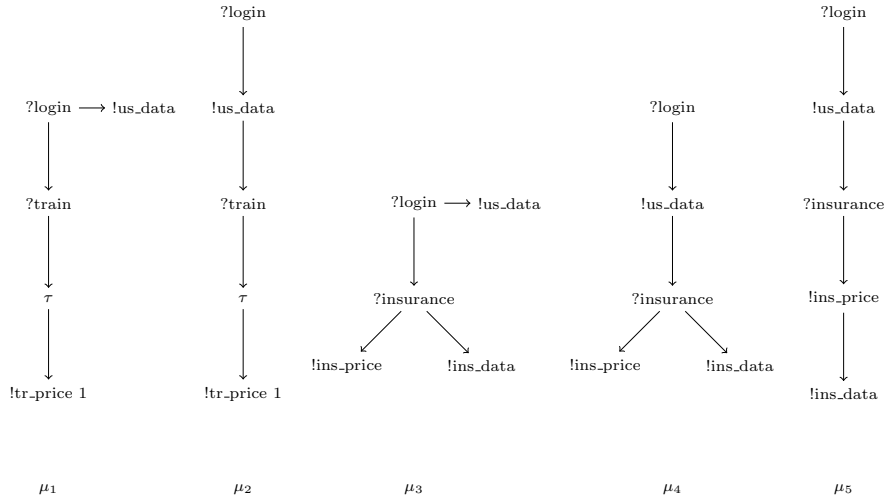


Fig. 4. Relaxed executions of the travel agency

Example 6. In Figure 4 we can see that μ_1 and μ_3 respect the structure of s and then both are relaxed executions of it ($\perp_s \xrightarrow{\mu_1}$ and $\perp_s \xrightarrow{\mu_3}$), but we have seen that $\mu_1 \sqsubseteq \mu_2, \mu_3 \sqsubseteq \mu_4$ and $\mu_3 \sqsubseteq \mu_5$, therefore μ_2, μ_4 and μ_5 are also relaxed executions of s ($\perp_s \xrightarrow{\mu_2}, \perp_s \xrightarrow{\mu_4}$ and $\perp_s \xrightarrow{\mu_5}$). We can conclude that the same structure can have several relaxed executions.

3 Observing Event Structures

The notion of conformance in a testing framework is based on the chosen notion of observation of the system behavior. One of the most popular ways of defining the behavior of a system is in terms of its *traces* (observable sequences of actions of the system). Phillips [8] proposes a conformance relation that in addition considers the actions that the system *refuses* when the communication is symmetric while Heerink and Tretmans [13] and Lestiennes and Gaudel [14] consider refusals with asymmetric communication. Finally, when there is a distinction between inputs and output actions, one can differentiate between situations where the system is still processing some information from those where the system can not evolve without the interaction of the environment (usually called *quiescence*). This notion was introduced by Segala in [10]. In this section, we define these three notions in the context of labeled event structures, since the **co-ioco** conformance relation that we present in Section 4 relies on these observations.

3.1 Traces

The labels in L represent the observable actions of a system; they model the system's interactions with its environment. Internal actions are denoted by the special label $\tau \notin L$. As explained in the previous section, the executions of a concurrent system are captured by pomsets, consequently, the observable behavior can be captured by abstracting the internal actions from the executions of the system.

Definition 10 (τ-abstraction of a pomset). *Let $\mu, \omega \in \mathcal{POMSET}(L)$, we have that $abs(\mu) = \omega$ iff there exist $lpo_\mu = (E_\mu, \leq_\mu, \lambda_\mu) \in \mu$ and $lpo_\omega = (E_\omega, \leq_\omega, \lambda_\omega) \in \omega$ such that*

- $E_\omega = \{e \in E_\mu \mid \lambda_\mu(e) \neq \tau\}$
- $\leq_\omega = \leq_\mu \cap (E_\omega \times E_\omega)$
- $\lambda_\omega = \lambda_{\mu_{|E_\omega}}$

Finally, an *observation* of a configuration is the τ-abstraction of one of its execution.

Definition 11 (Observation). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$, $\omega \in \mathcal{POMSET}(L)$ and $C, C' \in \mathcal{C}(\mathcal{E})$, we define*

$$C \stackrel{\omega}{\Longrightarrow} C' \triangleq \exists \mu : C \stackrel{\mu}{\longrightarrow} C' \text{ and } abs(\mu) = \omega$$
$$C \stackrel{\omega}{\Longrightarrow} \quad \triangleq \exists C' : C \stackrel{\omega}{\Longrightarrow} C'$$

We say that ω is an observation of C if $C \stackrel{\omega}{\Longrightarrow}$.

In the sequential framework, ? and ! are used to denote input and output actions respectively. We extend this and denote by $?\omega$ and $!\omega$ observations composed only by input and output actions respectively.

We can now define the notion of traces and reachable configurations from a given configuration by an observation. Our notion of traces is similar to the one of Ulrich and König [19] where a trace is considered as a sequence of partial orders. The reachable configurations that we consider are those that can be reached by abstracting the silent actions of an execution and only considering observable ones. This notion is similar to the one of unobservable reach proposed by Genc and Lafortune [27].

Definition 12 (Traces and reachable configurations). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$, $\omega \in \mathcal{POMSET}(L)$ and $C, C' \in \mathcal{C}(\mathcal{E})$, we define*

- $traces(\mathcal{E}) \triangleq \{\omega \in \mathcal{POMSET}(L) \mid \perp_{\mathcal{E}} \stackrel{\omega}{\Longrightarrow}\}$
- C ***after*** $\omega \triangleq \{C' \mid C \stackrel{\omega}{\Longrightarrow} C'\}$

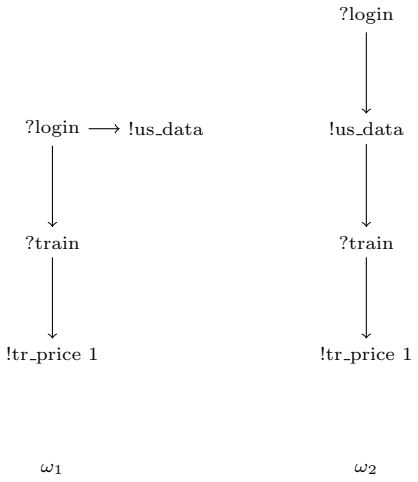


Fig. 5. Traces of the travel agency

Example 7. We can see that $abs(\mu_1) = \omega_1$ and $abs(\mu_2) = \omega_2$ and we saw in Example 6 that $\perp_s \xrightarrow{\mu_1}$ and $\perp_s \xrightarrow{\mu_2}$, therefore $\perp_s \overset{\omega_1}{\Longrightarrow}$ and $\perp_s \overset{\omega_2}{\Longrightarrow}$. It is also easy to see that $abs(\mu_3) = \mu_3, abs(\mu_4) = \mu_4$ and $abs(\mu_5) = \mu_5$, therefore $\perp_s \overset{\mu_3}{\Longrightarrow}, \perp_s \overset{\mu_4}{\Longrightarrow}$ and $\perp_s \overset{\mu_5}{\Longrightarrow}$. Finally $\omega_1, \omega_2, \mu_3, \mu_4, \mu_5 \in \text{traces}(s)$.

The configuration reached in s (from the initial configuration) after the observations μ_3, μ_4 and μ_5 is the same, i.e. $(\perp_s \textbf{ after } \mu_3) = (\perp_s \textbf{ after } \mu_4) = (\perp_s \textbf{ after } \mu_5) = \{\{?login, !us\_data, ?insurance, !ins\_price, !ins\_data\}\}$.

As there are unobservable actions and the labeling function may not be injective, the set of reachable configurations after a trace is in general not a singleton. A system where we can unambiguously detect the configuration reached after some observation is called *deterministic*.

Definition 13 (Deterministic IOLES). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$, we have*

$$\mathcal{E} \textit{ is deterministic} \Leftrightarrow \forall \omega \in \textit{traces}(\mathcal{E}), (\perp_{\mathcal{E}} \textbf{ after } \omega) \textit{ is a singleton}$$

Example 8. We can see in Figure 6 that there is not any distinction between the ticket choice, i.e. two configurations can be reached from the initial configuration after observing $?login \cdot ?ticket$, and then s_4 is not deterministic.

The non determinism of an event structure comes from at least one of the two following situations: two events in immediate conflict are labeled by the same action, i.e. $\exists e, e' \in E : e \overline{\#} e'$ and $\lambda(e) = \lambda(e')$; or two concurrent events are equally labeled, i.e. $\exists e, e' \in E : e \textbf{ co } e'$ and $\lambda(e) = \lambda(e')$. Because of Assumption 1, we only consider non determinism coming from immediate conflict.

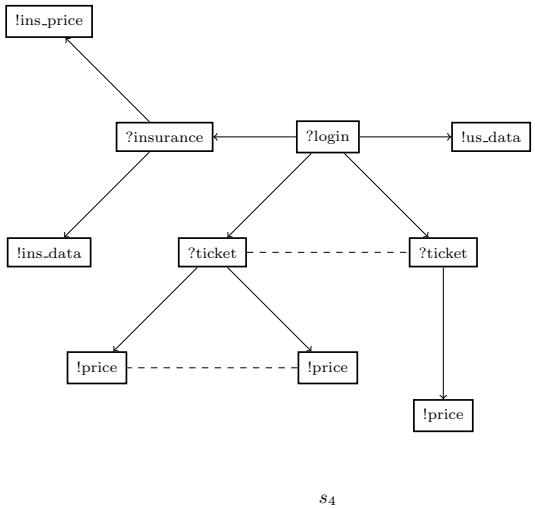


Fig. 6. A non deterministic IOLES.

3.2 Quiescence

With reactive systems, we need to differentiate configurations where the system can still produce some outputs and those were the system can not evolve without an input from the environment. Such situations are captured by the notion of quiescence [10]. The observation of quiescence in such configurations is usually instrumented by timers. Jard and Jéron [28] present three different kinds of quiescence: *output quiescence*: the system is waiting for an input from the environment, *deadlock*: the system can not evolve, and *livelock*: the system diverges by an infinite sequence of silent actions. The observation of quiescence is usually made explicit by adding self loops labeled by a δ action on quiescent states, where δ is considered as a new output action. But since event structures are acyclic, we denote the δ action semantically and not syntactically.

Definition 14 (Quiescence). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$ and $C \in \mathcal{C}(\mathcal{E})$, we have*

$$C \text{ is quiescent } \Leftrightarrow \forall\ !\omega \in \mathcal{POMSET}(L_o) : C \overset{!\omega}{\not\Longrightarrow}$$

We assume that we can observe quiescence by a δ action, i.e. if C is quiescent, then $C \overset{\delta}{\Longrightarrow}$.

Both output quiescence and deadlock are captured by the definition above, while livelock is not possible by Assumption 2.

Example 9. In the travel agency example, the configuration reached after logging in is not quiescent as the user's data can be sent, i.e $(\perp_s \textbf{ after } ?login) = \{\{?login\}\}$ and $\{?login\} \overset{!us\_data}{\Longrightarrow}$, but the configuration reached after sending

the user's data is quiescent because only input actions are enabled, i.e. $(\perp_s$ **after** $(?login \cdot !us\_data)) = \{\{?login, !us\_data\}\}$ and for every $\omega = (E_\omega, \leq_\omega, \lambda_\omega)$ such that $\{?login, !us\_data\} \stackrel{\omega}{\Longrightarrow}$, we have $E^{\mathcal{I}}_\omega \neq \emptyset$.

In the LTS framework, the *produced outputs* of the systems are single actions. A first extension proposed by the authors in [22] considers that the outputs produced by the system in response to stimuli could be single actions as well as sets of concurrent ones. However, as we allow concurrent outputs to be ordered, we will consider the outputs produced by a system as the maximal pomsets of outputs that can be produced in the current configuration, i.e. the configuration reached after such outputs is quiescent. Such configuration always exists by Assumption 2. As usual, a δ action is an special observation that allows to detect quiescent configurations.

Definition 15 (Produced outputs). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$ and $S \subseteq \mathcal{C}(\mathcal{E})$, we define*

$$out(S) \triangleq \bigcup_{C \in S} \{!\omega \in \mathcal{POMSET}(L_o) \mid C \stackrel{!\omega}{\Longrightarrow} C' \ \wedge \ C' \text{ is quiescent}\}$$

$$\cup \{\delta \mid \exists C \in S : C \stackrel{\delta}{\Longrightarrow}\}$$

Example 10. The only output produced by the system of Figure 2 after logging in is the user's data, i.e. out($\perp_s$ **after** $?login) = \{!us\_data\}$ and after this output, a quiescent configuration is reached, then out($\perp_s$ **after** $(?login \cdot !us\_data)) = \{\delta\}$. If a train ticket is chosen, different prices may be produced, i.e. out($\perp_s$ **after** $(?login \cdot !us\_data \cdot ?train)) = \{!tr\_price\ 1, !tr\_price\ 2\}$. However, due to the possible ordering between concurrent outputs, the outputs produced by the implementation after choosing the insurance can be observed in different ways as it is shown in the traces of Figure 5, out($\perp_s$ **after** $(?login \cdot !us\_data \cdot ?insurance))$ $= \{!ins\_price$ **co** $!ins\_data, !ins\_price \cdot !ins\_data, !ins\_data \cdot !ins\_price\}$.

3.3 Refusals

The **ioco** theory assumes the input enableness of the implementation, i.e. in any state of the implementation, every input action is enabled. This assumption is made to avoid computation interference [29] in the parallel composition between the implementation and the test cases. However, as explained by Heerink, Lestiennes and Gaudel in [30, 14] 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. An example of such a system is an automatic cash dispenser where the action of introducing a card becomes (physically) unavailable after inserting a card, as the automatic cash dispenser is not able to swallow more than one card at a time. Furthermore, this theory proposes test cases that are always capable of observing every output produced by the system, a non very realistic situation in an concurrent environment.

In order to overcome these difficulties, Heerink [30] distributes the points of control and observation, and the input enableness assumption is weakened by the following assumption: "if an input action can be performed in a control point, all the inputs actions of that control point can be performed". Refused inputs in the implementation are made observable by an special ξ-action (as quiescence is observable by a δ action). In [14], 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.

Our approach is closer to [14]; any *possible input* in a configuration of the specification should also be possible in the implementation (or any input refused by the implementation should be refused by the specification). In an observation, an input action may be preceded by an output that was specified to be concurrent (as it is the case with $!us\_data$ and $?train$ in ω_2); therefore $?train$ should still be consider as possible even if the $!us\_data$ output hasn't been produced yet. The possible inputs of a configuration are those that are enabled or those that will be enabled after producing some outputs.

Definition 16 (Possible inputs). *Let $\mathcal{E} \in \mathcal{IOLES}(L)$ and $S \subseteq \mathcal{C}(\mathcal{E})$, we define*

$$poss(S) \triangleq \bigcup_{C \in S} \{?\omega \in \mathcal{POMSET}(L_i) \mid C \stackrel{?\omega}{\Longrightarrow} \ \vee \ \exists\, !\omega \in \mathcal{POMSET}(L_o) : C \stackrel{!\omega}{\Longrightarrow} C' \ \wedge \ C' \stackrel{?\omega}{\Longrightarrow}\}$$

Example 11. If we consider the IOLES i_1 of Figure 7, we see that the first possible input is the logging in, i.e. $poss(\perp_{i_1}) = \{?login\}$. After the user has logged in, the selections become possible even before the user's data is sent. These selections are possible as individual actions or as concurrent ones, i.e. $poss(\perp_{i_1}$ **after** $?login) = \{?insurance, ?train, ?plane, ?insurance$ **co** $?train, ?insurance$ **co** $?plane\}$.

In order to allow the observation of the possible inputs of the system under test, a configuration where inputs are possible should not allow alternatively the production of outputs. As a matter of fact, if an input and an output are in conflict in a given configuration, once the output is produced, the input is not enabled anymore. Such configurations prevent from observing the possible inputs of the system under test. For this reason, we restrict the form of labeled event structures we consider with the following assumption.

Assumption 3 *We will only consider IOLES such that there is no immediate conflict between input and output events, i.e. $\forall e \in E^{\mathcal{I}}, e' \in E^{\mathcal{O}} : \neg(e \ \overline{\#} \ e')$.*

This assumption is also made by Gaudel et al in [14], where such specifications are called IO-exclusive. With such an assumption, we can prove that any enabled input can not be disabled by the production of some outputs.

Proposition 1. *If we have an IOLES such that there is no immediate conflict between an input event and an output one, then any possible input in a non quiescent configuration is possible after producing the corresponding outputs.*

Proof. Let assume that $?\omega \in \text{poss}(C)$, then either $C \overset{?\omega}{\Longrightarrow}$, or we can reach from C a quiescent configuration C' such that $C' \overset{?\omega}{\Longrightarrow}$. If it is the second case, the result is immediate. If it is the case that $C \overset{?\omega}{\Longrightarrow}$, let C'' be the quiescent configuration reachable from it by some $!\omega$, i.e. $C \overset{!\omega}{\Longrightarrow} C''$ and C'' is quiescent. We have two observations $?\omega$ and $!\omega$ that are possible at the same configuration C and then its events should be in immediate conflict or concurrent. By the hypothesis, we know they are concurrent (they are not in immediate conflict) and then they remain observable after some of them have been observed, i.e. $C \overset{!\omega}{\Longrightarrow} C''$ and $C'' \overset{?\omega}{\Longrightarrow}$. Finally $?\omega \in \text{poss}(C'')$.

We now have all the elements to define a conformance relation for IOLES based in the observation notions of traces, refusals and quiescence.

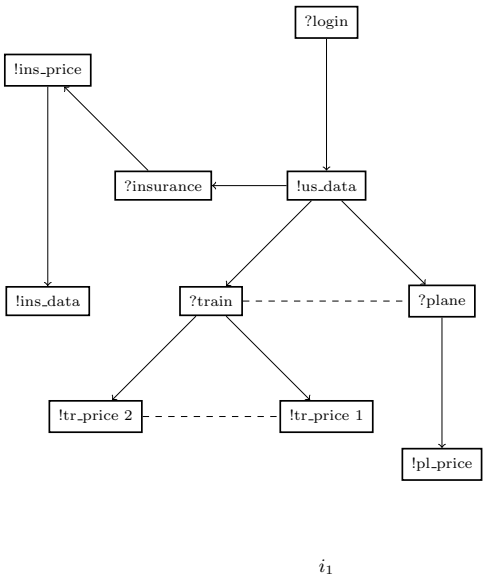


Fig. 7. A correct implementation w.r.t **co-ioco** of the travel agency of Figure 2.

4 The conformance relation: co-ioco

The activity of testing relies crucially on the definition of a *conformance* relation that specifies which observed behaviors must be considered conforming,

or *not* conforming, to the specification. Aceto et al [26] propose several testing equivalences depending in the chosen semantics for event structures. In [22], we propose two extensions for the **ioco** conformance relation, one for the interleaving semantics and another for the partial order one.

We refine here the second conformance relation defined in [22]. We define a conformance relation for labeled event structures with the partial order semantics that extends the **ioco** conformance relation proposed by Tretmans [31]. Informally, an implementation i conforms to specification s for **ioco** if any experiment derived from s and executed on i leads to an output from i that is foreseen by s. A special output of i is the absence of outputs (quiescence), modeled by a δ action. This means that if i is quiescent after the experiment, then s should have the possibility to be quiescent after it too.

Our conformance relation for labeled event structure 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.

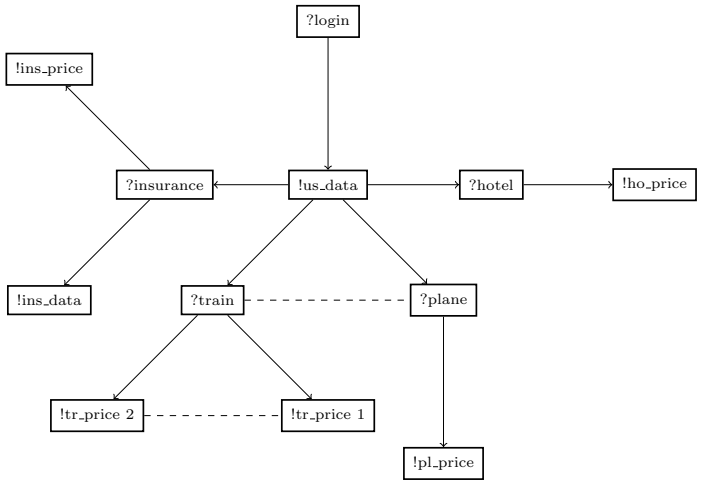
Definition 17 (co-ioco). *Let $i, s \in \mathcal{IOLES}(L)$, then*

$$\begin{aligned} i \ \textbf{\textit{co-ioco}}\ s \Leftrightarrow \forall \omega \in \mathit{traces}(s) : \\ \mathit{poss}(\perp_s \textbf{\textit{after}}\ \omega) \subseteq \mathit{poss}(\perp_i \textbf{\textit{after}}\ \omega) \\ \mathit{out}(\perp_i \textbf{\textit{after}}\ \omega) \ \subseteq \ \mathit{out}(\perp_s \textbf{\textit{after}}\ \omega) \end{aligned}$$

It is not assumed here that the implementation is input enabled, but if one makes such an assumption, the **co-ioco** definition boils down to the inclusion of produced outputs as it is the case in the **ioco** relation.

Example 12 (Ordered Outputs). The implementation i_1 of the travel agency proposed in Figure 7 order some output actions, for example $!us\_data \leq ?insurance$ instead of $!us\_data$ **co** $?insurance$ and $!ins\_price \leq !ins\_data$ instead of $!ins\_price$ **co** $!ins\_data$, but all the possible inputs of the specification are implemented (respecting their concurrency) and every output/quiescence produced by the implementation is specified. We can conclude that i_1 **co-ioco** s.

Example 13 (Extra Inputs). The behavior of the implementation and the specification is compared after some observations are made. These observations are taken from the specification (they are traces of s), therefore, there is no restriction for the implementation about how to react to unspecified inputs. Figure 8 shows a possible implementation i_2 that also allows the user to make a hotel reservation ($?hotel$). Even if this implementation may produce an extra output ($!ho\_price$), this output is only produced after the hotel has been chosen, but the behavior of the system after choosing a hotel is not specified so i_2 **co-ioco** s.



i_2

Fig. 8. Extra inputs.

Example 14 (Refused Inputs). The conformance relation considers the input actions that the implementation may refuse. Figure 9 presents two possible implementations i_3 and i_4 of the travel agency. The one on the left removes the possibility to choose an insurance while the one on the right removes the choice for a train ticket. In the specification, we have that $\text{poss}(\bot_s \textbf{ after } ?login) = \{?insurance, ?train, ?plane, ?insurance \textbf{ co } ?train, ?insurance \textbf{ co } ?plane\}$, but $?insurance$ is not possible in i_3, i.e. $?insurance \notin \text{poss}(\bot_{i_3} \textbf{ after } ?login) = \{?train, ?plane\}$ and finally $\neg(i_3 \textbf{ co-ioco } s)$. The $?train$ action is neither possible in i_4, i.e. $?train \notin \text{poss}(\bot_{i_4} \textbf{ after } ?login) = \{?insurance, ?plane, ?insurance \textbf{ co } ?plane\}$ and then $\neg(i_4 \textbf{ co-ioco } s)$.

Example 15 (Extra/incomplete Outputs). The second condition of the conformance relation establishes that all the outputs produced by the implementation should by specified. Consider the implementation i_5 presented in Figure 10: after choosing the plane ticket, the implementation can produce an output with the ticket's price or an error message due to the fact that there are not tickets available, i.e. $\text{out}(\bot_{i_5} \textbf{ after } (?login \cdot !us\_data \cdot ?plane)) = \{!pl\_price, !pl\_full\}$, but $!pl\_full$ is not a possible output in the specification, i.e. $!pl\_full \notin \text{out}(\bot_s \textbf{ after } (?login \cdot !us\_data \cdot ?plane)) = \{!pl\_price\}$, therefore $\neg(i_5 \textbf{ co-ioco } s)$. The conformance relation only considers "complete" outputs (those that lead to a quiescent configuration), while incomplete outputs lead to non conformance. Implementation i_5 also shows an example of this: after choosing the insurance, only its price is produced, i.e. $\text{out}(\bot_{i_5} \textbf{ after } (?login \cdot !us\_data \cdot ?insurance)) = \{!ins\_price\}$, while it is specified that some data should also be produced (con-

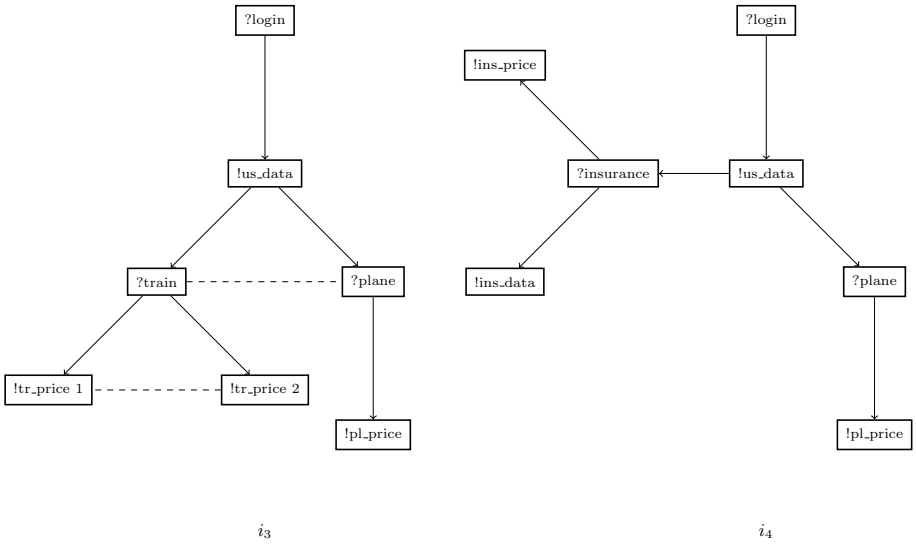


Fig. 9. Refused inputs.

currently or in some order), i.e. $\text{out}(\bot_s \textbf{ after } (?login \cdot !us\_data \cdot ?insurance)) = \{!ins\_price \textbf{ co } !ins\_data, !ins\_price \cdot !ins\_data, !ins\_data \cdot !ins\_price\}$, and again $\neg(i_5 \textbf{ co-ioco } s)$.

Example 16 (Extra Quiescence). The second condition stipulates that the absence of outputs can only occur when it is specified. Figure 11 shows an implementation i_6 that does not send the user's data after logging in, thus a quiescent configuration is reached after logging in, i.e. $\text{out}(\bot_{i_6} \textbf{ after } ?login) = \{\delta\}$, but this quiescence is not specified, i.e. $\delta \notin \text{out}(\bot_s \textbf{ after } ?login) = \{!us\_data\}$, and $\neg(i_7 \textbf{ co-ioco } s)$.

5 Testing from Labeled Event Structures

In section 4 we have formally defined what it means for an implementation to conform to its specification and we have seen several examples of conforming and non conforming implementations. Now we need a way to test this notion of conformance. In this section we define the notion of test cases, test suites, their interaction with the implementations and we give sufficient conditions for detecting all and only incorrect implementations. We finally give an algorithm to generate test suites verifying these conditions.

In order to formally reason about implementations, we assume as it is usual that any real implementation can be modeled by some formal object, in our case, an IOLES.

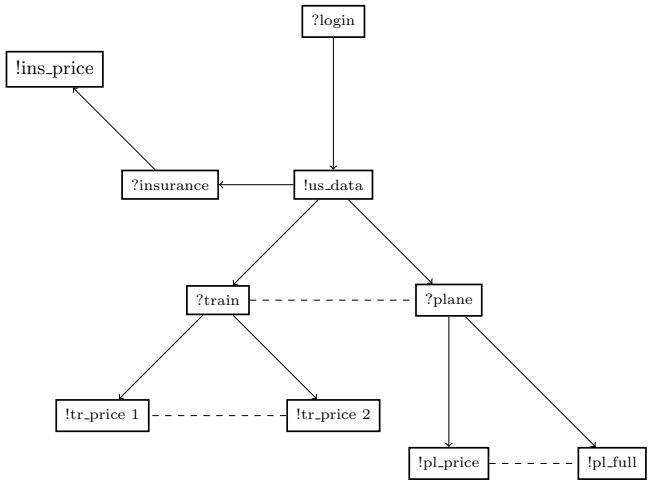


Fig. 10. Extra/Incomplete Outputs.

5.1 Test Cases and Test Suites

A *test case* is a specification of the tester's behavior during an experiment carried out on the system under test. In such an experiment, the tester serves as a kind of artificial environment of the implementation. The output[3] actions are not controlled by the tester, but the input ones are, and therefore there should not be choices between them, i.e. the next (set of concurrent) input(s) to be proposed should be unique, therefore there should not be immediate conflict between inputs.

This property is not enough to avoid the choices in a test case: if we allow the tester to reach more than one configuration after some observation and each of them enables different inputs, there is still some (non deterministic) choice for the tester about the next input to propose even if those inputs are not in immediate conflict, therefore the reached configuration after some observation should be unique.

Finally, we require the experiment to finish, therefore the event structure should be finite.

We model the behavior of the tester by a deterministic event structure with a finite set of events and without immediate conflicts between its inputs.

Definition 18 (Test Case / Test Suite). *A test case is a deterministic input/output labeled event structure* $t = (E_t, \leq_t, \#_t, \lambda_t)$ *where*

[3] When we refer to inputs/outputs, we refer to input or output from the point of view of the implementation. We do not assume, as it is usual, that the test case is a "mirror" of the specification

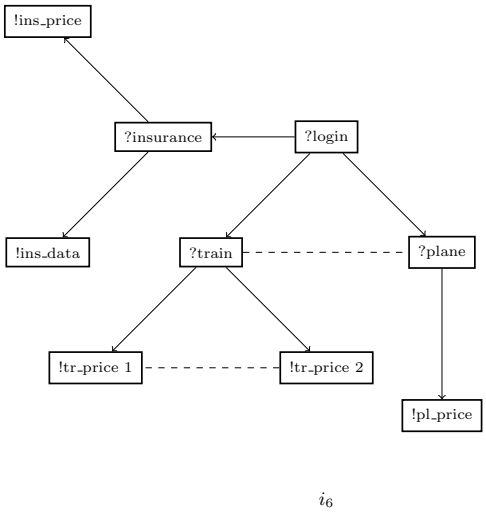


Fig. 11. Extra Quiescence.

1. $(E_t^{\mathcal{I}} \times E_t^{\mathcal{I}}) \cap \overline{\#}_t = \emptyset$,
2. E_t *is finite*

A test suite *is a set of test cases.*

Example 17. Figure 12 presents three event structures. The behavior of t_1 is infinite which prevents it from being a test case; t_2 is not a test case either since there is an immediate conflict between in_2 and in_3. The inputs in_3 and in_4 are in conflict in t_3, but this conflict is not immediate. In addition E_{t_3} is finite and t_3 is deterministic, therefore t_3 is a test case.

We are interested in the interaction between the test case and the implementation (called *test execution*) in order to give a *verdict* about the success or failure of the test w.r.t. the conformance relation. Verdicts are usually modeled via a labeling function from the states of the test case to the set $\{\mathbf{pass}, \mathbf{fail}\}$. Only leaves are labeled and a pass verdict can only be reached after observing some output of the implementation (in this framework, δ is considered an output). One possibility would be to label configurations with verdicts, but as there is no event labeled by δ, i.e. observing δ does not lead to a new configuration, we need to model verdicts differentely. As in the case of quiescence, we do not define verdicts syntactivally, but rather semantically.

5.2 Test Execution and Verdicts

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

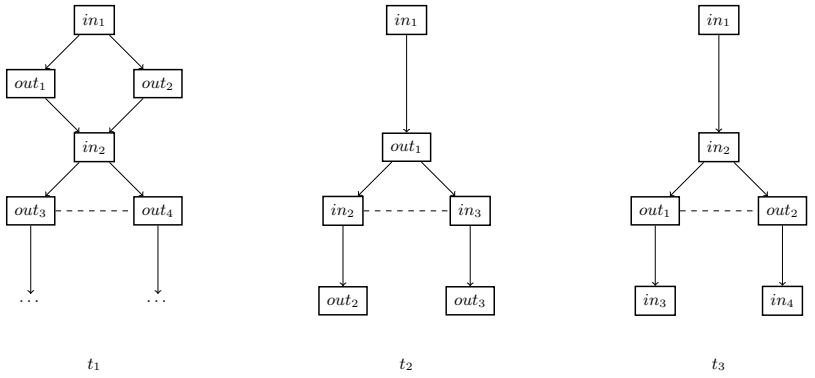


Fig. 12. Left: an infinite event structure, **Center**: immediate conflict between inputs, **Right**: a test case

assumed that the implementation accepts any input the tester can proposed (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). 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/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 ω, the test execution can block because of an output the implementation produces for two reasons. First, if after such an observation the test case can not accept that output:

$$\exists!\omega \in \mathcal{POMSET}(L_o), C_i \in (\perp_i \textbf{ after } \omega) : C_i \stackrel{!\omega}{\Longrightarrow} \text{ and}$$

$$\forall C_t \in (\perp_t \textbf{ after } \omega) : C_t \stackrel{!\omega}{\not\Longrightarrow}$$

Second the test case can accept such output, but this is not the maximal output it can accept (the reached configuration is not quiescent):

$$C_t \stackrel{!\omega}{\Longrightarrow} \text{ implies } (\exists!\omega' \in \mathcal{POMSET}(L_o) : \ E_{!\omega} \subset E_{!\omega'} \text{ and } C_t \stackrel{!\omega'}{\Longrightarrow})$$

We will consider the set of outputs leading the implementation to a quiescent configuration. Both situation can be simplified to the observation of an element

in the set of outputs produced by the implementation that is not in the set of outputs of the test case.

Definition 19 (Blocking because of an output). *Let $i, t \in \mathcal{IOLES}(L)$ and $\omega \in \mathcal{POMSET}(L)$, we have*

$$\textbf{\textit{blocks}}_{\mathcal{O}}(i,t,\omega) \Leftrightarrow \exists!\omega \in \mathcal{POMSET}(L_o) : !\omega \in out(\perp_i \textbf{\textit{after}}\ \omega) \text{ and } !\omega \notin out(\perp_t \textbf{\textit{after}}\ \omega)$$

Example 18. Consider the implementation i_5, the test case t_4 presented in Figure 13 and let $\omega_1' = (?login \cdot !us\_data \cdot ?plane)$. We have that the test execution blocks after ω_1' because the implementation produces a $!pl\_full$ action (which leads to a quiescent configuration) and the test case is not able to accept it, i.e. $!pl\_full \in \text{out}(\perp_{i_5} \textbf{after}\ \omega_1')$, but $!pl\_full \notin \text{out}(\perp_{t_4} \textbf{after}\ \omega_1')$, and finally $\textbf{blocks}_{\mathcal{O}}(i_5, t_4, \omega_1')$. If we consider $\omega_2' = (?login \cdot !us\_data \cdot ?insurance)$, the test execution also blocks because the $!ins\_price$ action proposed by the implementation (leading to a quiescent configuration) is enabled in the test case, but the reached configuration is not quiescent because $!ins\_data$ is still enabled, i.e. $!ins\_price \in \text{out}(\perp_i \textbf{after}\ \omega_2'), !ins\_price \notin \text{out}(\perp_t \textbf{after}\ \omega_2')$, and $\textbf{blocks}_{\mathcal{O}}(i_5, t_4, \omega_2')$.

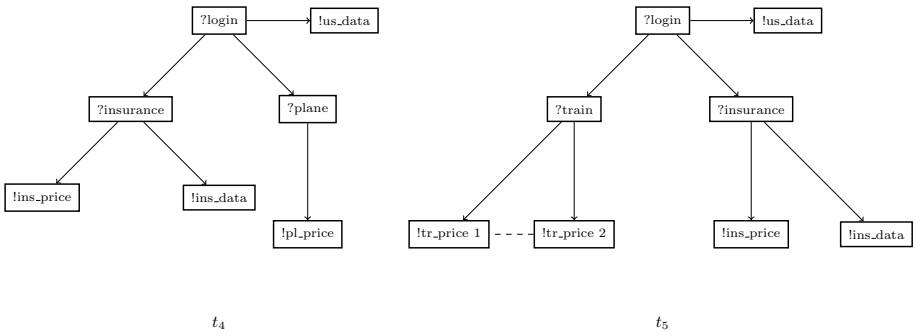


Fig. 13. Two test cases for the travel agency in Figure 2

Another blocking situation is when a quiescent configuration is reached in the implementation (observed by a δ action) but the δ action is not observable in the test case.

Definition 20 (Blocking because of quiescence). *Let $i, t \in \mathcal{IOLES}(L)$ and $\omega \in \mathcal{POMSET}(L)$, we have*

$$\textbf{\textit{blocks}}_{\delta}(i,t,\omega) \Leftrightarrow \delta \in out(\perp_i \textbf{\textit{after}}\ \omega) \text{ and } \delta \notin out(\perp_t \textbf{\textit{after}}\ \omega)$$

Example 19. Consider the implementation i_6 and the test case t_4. We have that the test execution blocks after $?login$ because the implementation reaches a quiescent configuration, i.e. $(\bot_{i_6}$ **after** $?login) \stackrel{\delta}{\Longrightarrow}$ and $\delta \in \text{out}(\bot_{i_6}$ **after** $?login)$, but δ is not observable in the test case, i.e. $\delta \notin \text{out}(\bot_{t_4}$ **after** $?login)$, and $\mathbf{blocks}_{\delta}(i_6, t_4, ?login)$.

The last blocking situation happens when the test case can propose an input that the implementation is not prepared to accept, i.e. $\exists C_t \in (\bot_t$ **after** $\omega), ?\omega \in \mathcal{POMSET}(L_i) : C_t \stackrel{?\omega}{\Longrightarrow}$ and $\forall C_i \in (\bot_i$ **after** $\omega) \stackrel{?\omega}{\not\Longrightarrow}$. As the implementation can add some causality depending on an output action, we should also consider the inputs that will become enabled after producing some outputs.

Definition 21 (Blocking because of an input). *Let $i, t \in \mathcal{IOLES}(L)$ and $\omega \in \mathcal{POMSET}(L)$, we have*

$$\begin{aligned}\boldsymbol{blocks}_{\mathcal{I}}(i, t, \omega) \Leftrightarrow \exists ?\omega \in \mathcal{POMSET}(L_i) : ?\omega \in \mathit{poss}(\bot_t \ \boldsymbol{after}\ \omega) \ \mathit{and} \\ ?\omega \notin \mathit{poss}(\bot_i \ \boldsymbol{after}\ \omega)\end{aligned}$$

Example 20. Consider the implementation i_3 and the test case t_5 of Figure 13. The test execution blocks after logging in because the test case can propose an $?insurance$ action, i.e. $?insurance \in \text{poss}(\bot_{t_5}$ **after** $?login)$, that the implementation is not able to accept it, i.e. $?insurance \notin \text{poss}(\bot_{i_3}$ **after** $?login)$, and $\mathbf{blocks}_{\mathcal{I}}(i_3, t_5, ?login)$. If we consider i_4 as the implementation, the test execution also blocks because the $?train$ input action proposed by the test case, i.e. $?train \in \text{poss}(\bot_{t_5}$ **after** $?login)$, is not possible in the implementation $?train \notin \text{poss}(\bot_{i_4}$ **after** $?login)$, and $\mathbf{blocks}_{\mathcal{I}}(i_4, t_5, ?login)$.

We can now define the verdict of the executions of a set a test cases with an implementation.

Definition 22 (Failure of a test suite). *Let i be an implementation, and T a test suite, we have:*

$$\begin{aligned}i \ \mathbf{fails}\ T \Leftrightarrow \exists t \in T, \omega \in \mathit{traces}(t) : \boldsymbol{blocks}_{\mathcal{O}}(i, t, \omega) \\ \vee \boldsymbol{blocks}_{\delta}(i, t, \omega) \vee \boldsymbol{blocks}_{\mathcal{I}}(i, t, \omega)\end{aligned}$$

If the implementation does not fail the test suite, it passes *it.*

Example 21. Let $T = \{t_4, t_5\}$ from Figure 13. We have seen in section 4 several situations that lead to the non conformance of an implementation. As seen in Examples 18 and 19, the execution of the test case t_4 with the (non conforming) implementations i_5, i_6 leads to a blocking, so we have i_5, i_6 **fails** T. We saw in Example 20 that the test executions between the implementations i_3, i_4 and the test case t_5 block after logging in, therefore we also have i_3, i_4 **fails** T. We can conclude that T is capable of detect the non conforming implementations presented in the last section. We let the reader check that the (correct) implementations i_1, i_2 pass T.

5.3 Completeness of the Test Suite

We saw in Example 21 that all the possible situations seen in Section 4 that may lead to the non conformance of the implementation are detected by the test suite $\{t_4, t_5\}$. When testing implementations, we intend to detect all, and only all, non conforming implementations. This notion is formalized by the *completeness* definition. Complete test suite are usually infinitely large, and not executable in a practical situation. Consequently, a distinction is made between test suites which detect only errors - but possibly not all of them - and test suites which detect all errors - and possible more. The former are called *sound*, and the latter *exhaustive*.

Definition 23 (Complete / Sound / Exhaustive test suite). *Let s be a specification and T a test suite, then*

$$\begin{array}{lll} T \textit{ is complete} & \triangleq \forall i : i \textbf{ fails } T & \textit{iff} \quad \neg(i \textbf{ co-ioco } s) \\ T \textit{ is sound} & \triangleq \forall i : i \textbf{ fails } T & \textit{implies} \quad \neg(i \textbf{ co-ioco } s) \\ T \textit{ is exhaustive} & \triangleq \forall i : i \textbf{ fails } T & \textit{if} \quad \neg(i \textbf{ co-ioco } s) \end{array}$$

The following theorem gives sufficient conditions for having a sound test suite.

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

a) $\forall t \in T : traces(t) \subseteq traces(s)$
b) $\forall t \in T, \omega \in traces(t) : out(\bot_t \textbf{ after } \omega) = out(\bot_s \textbf{ after } \omega)$
c) $\forall t \in T, \omega \in traces(t) : poss(\bot_t \textbf{ after } \omega) \subseteq poss(\bot_s \textbf{ after } \omega)$

then T is sound for s w.r.t **co-ioco**.

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 22 we have:

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

and at least one of the following cases holds:

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

$$\begin{array}{ll} & \exists \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 19 } *\} \\ & \exists \omega \in \text{traces}(t), !\omega \in \mathcal{POMSET}(L_o) : !\omega \in \text{out}(\bot_i \textbf{ after } \omega) \text{ and} \\ & !\omega \notin \text{out}(\bot_t \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Hypotheses a) and b) } *\} \\ & \exists \omega \in \text{traces}(s) : \text{out}(\bot_i \textbf{ after } \omega) \not\subseteq \text{out}(\bot_s \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 17 } *\} \\ & \neg(i \textbf{ co-ioco } s) \end{array}$$

2. the test execution blocks after ω because a δ action observed in the implementation:

$$\begin{array}{ll} & \exists \omega \in \text{traces}(t) : \mathbf{blocks}_{\delta}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 20 } *\} \\ & \exists \omega \in \text{traces}(t) : \delta \in \text{out}(\bot_i \textbf{ after } \omega) \text{ and } \delta \notin \text{out}(\bot_t \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Hypotheses a) and b) } *\} \\ & \exists \omega \in \text{traces}(s) : \text{out}(\bot_i \textbf{ after } \omega) \not\subseteq \text{out}(\bot_s \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 17 } *\} \\ & \neg(i \textbf{ co-ioco } s) \end{array}$$

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

$$\begin{array}{ll} & \exists \omega \in \text{traces}(t) : \mathbf{blocks}_{\mathcal{I}}(i,t,\omega) \\ \text{implies} & \{* \text{ Definition 21 } *\} \\ & \exists \omega \in \text{traces}(t), ?\omega \in \mathcal{POMSET}(L_i) : ?\omega \in \text{poss}(\bot_t \textbf{ after } \omega) \text{ and} \\ & ?\omega \notin \text{poss}(\bot_i \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Hypotheses a) and c) } *\} \\ & \exists \omega \in \text{traces}(s) : \text{poss}(\bot_s \textbf{ after } \omega) \not\subseteq \text{poss}(\bot_i \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 17 } *\} \\ & \neg(i \textbf{ co-ioco } s) \end{array}$$

We encourage the interested reader to check that the test suite $\{t_4, t_5\}$ proposed in Figure 12 is sound w.r.t the specification s.

The following theorem gives the sufficient conditions for the exhaustiveness of a test suite.

Theorem 2. *Let $s \in \mathcal{IOLES}(L)$ and T a test suite such that*

a) $\forall \omega \in traces(s) : (\exists t \in T : \omega \in traces(t))$

b) $\forall t \in T, \omega \in traces(t) : (\exists C_t \in (\bot_t \textbf{ after } \omega) : C_t \stackrel{\delta}{\Longrightarrow})$ *implies* $\exists C_s \in (\bot_s \textbf{ after } \omega) : C_s$ *is quiescent*

then T is exhaustive for 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 three cases holds:

1. $\exists \omega \in \text{traces}(s), !\omega \in \mathcal{POMSET}(L_o) : !\omega \in \text{out}(\bot_i \textbf{ after } \omega)$ and $!\omega \notin \text{out}(\bot_s \textbf{ after } \omega)$

$$\begin{array}{ll} \text{implies} & \{* \text{ Definition 15 } *\} \\ & \exists \omega \in \text{traces}(s), !\omega \in \mathcal{POMSET}(L_o) : \\ & \exists C_i \in (\bot_i \textbf{ after } \omega) : C_i \stackrel{!\omega}{\Longrightarrow} C_i' \text{ and } C_i' \text{ is quiescent and} \\ & (\forall C_s \in (\bot_s \textbf{ after } \omega) : C_s \stackrel{!\omega}{\not\Longrightarrow} \text{ or } (\forall C_s' : C_s \stackrel{!\omega}{\Longrightarrow} C_s' \text{ and} \\ & C_s' \text{ is not quiescent})) \end{array}$$

a) if the implementation produces an output that the specification can not produce:

$$\exists \omega \in \text{traces}(s), !\omega \in \mathcal{POMSET}(L_o) :$$
$$\exists C_i \in (\perp_i \textbf{ after } \omega) : C_i \stackrel{!\omega}{\Longrightarrow} C_i' \text{ and } C_i' \text{ is quiescent and}$$
$$\forall C_s \in (\perp_s \textbf{ after } \omega) : C_s \stackrel{!\omega}{\not\Longrightarrow}$$

Let t be such that $\omega \in \text{traces}(t)$ and $\forall C_t \in (\perp_t \textbf{ after } \omega) : C_t \stackrel{!\omega}{\not\Longrightarrow}$, such t exists by Hypothesis a)

$$\exists \omega \in \text{traces}(t), !\omega \in \mathcal{POMSET}(L_o) :$$
$$\exists C_i \in (\perp_i \textbf{ after } \omega) : C_i \stackrel{!\omega}{\Longrightarrow} C_i' \text{ and } C_i' \text{ is quiescent and}$$
$$\forall C_t \in (\perp_t \textbf{ after } \omega) : C_t \stackrel{!\omega}{\not\Longrightarrow}$$
implies $\{* \text{ Definition 15 } *\}$
$$\exists \omega \in \text{traces}(t), !\omega \in \mathcal{POMSET}(L_o) :$$
$$!\omega \in \text{out}(\perp_i \textbf{ after } \omega) \text{ and } !\omega \notin \text{out}(\perp_t \textbf{ after } \omega)$$
implies $\{* \text{ Definition 19 } *\}$
$$\exists \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i, t, \omega)$$
implies $\{* \text{ Definition 22 } *\}$
$$i \textbf{ fails } T$$

b) if the implementation produces an output that the specification produces but this output does not take the specification to a quiescent configuration:

$$\exists \omega \in \text{traces}(t), !\omega \in \mathcal{POMSET}(L_o) :$$
$$\exists C_i \in (\perp_i \textbf{ after } \omega) : C_i \stackrel{!\omega}{\Longrightarrow} C_i' \text{ and } C_i' \text{ is quiescent and}$$
$$\forall C_s \in (\perp_s \textbf{ after } \omega), C_s' : C_s \stackrel{!\omega}{\Longrightarrow} C_s' \text{ and } C_s' \text{ is not quiescent}$$

Let t be such that $(\omega \cdot !\omega) \in \text{traces}(t)$, by Hypothesis a), and $\forall C_t \in (\perp_t \textbf{ after } (\omega \cdot !\omega)) : C_t \stackrel{\delta}{\not\Longrightarrow}$, by Hypothesis b), we have

$$\exists (\omega \cdot !\omega) \in \text{traces}(t) :$$
$$\exists C_i' \in (\perp_i \textbf{ after } (\omega \cdot !\omega)) : C_i' \stackrel{\delta}{\Longrightarrow} \text{ and}$$
$$\forall C_t' \in (\perp_t \textbf{ after } (\omega \cdot !\omega)) : C_t \stackrel{\delta}{\not\Longrightarrow}$$
implies $\{* \text{ Definition 15 } *\}$
$$\exists (\omega \cdot !\omega) \in \text{traces}(t) :$$
$$\delta \in \text{out}(\perp_i \textbf{ after } (\omega \cdot !\omega)) \text{ and } \delta \notin \text{out}(\perp_t \textbf{ after } (\omega \cdot !\omega))$$
implies $\{* \text{ Definition 20 } *\}$
$$\exists (\omega \cdot !\omega) \in \text{traces}(t) : \textbf{blocks}_{\delta}(i, t, \omega \cdot !\omega)$$
implies $\{* \text{ Definition 22 } *\}$
$$i \textbf{ fails } T$$

2. $\exists \omega \in \text{traces}(s) : \delta \in \text{out}(\perp_i \textbf{ after } \omega) \text{ and } \delta \notin \text{out}(\perp_s \textbf{ after } \omega)$

implies $\{* \text{ Definition 15 } *\}$
$$\exists \omega \in \text{traces}(s) : \exists C_i \in (\perp_i \textbf{ after } \omega) : C_i \stackrel{\delta}{\Longrightarrow} \text{ and}$$
$$\forall C_s \in (\perp_s \textbf{ after } \omega) : C_s \stackrel{\delta}{\not\Longrightarrow}$$

Let t be such that $\omega \in \text{traces}(t)$, by Hypothesis a), and $\forall C_t \in (\bot_t \textbf{ after } \omega) : C_t \stackrel{\delta}{\not\Longrightarrow}$, by Hypothesis b), we have

$$
\begin{array}{ll}
 & \exists \omega \in \text{traces}(t) : \exists C_i \in (\bot_i \textbf{ after } \omega) : C_i \stackrel{\delta}{\Longrightarrow} \text{ and} \\
 & \forall C_t \in (\bot_t \textbf{ after } \omega) : C_t \stackrel{\delta}{\not\Longrightarrow} \\
\text{implies} & \{* \text{ Definition 15 } *\} \\
 & \exists \omega \in \text{traces}(t) : \delta \in \text{out}(\bot_i \textbf{ after } \omega) \text{ and } \delta \notin \text{out}(\bot_t \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 20 } *\} \\
 & \exists \omega \in \text{traces}(t) : \textbf{blocks}_{\delta}(i, t, \omega) \\
\text{implies} & \{* \text{ Definition 22 } *\} \\
 & i \textbf{ fails } T
\end{array}
$$

3. $\exists \omega \in \text{traces}(s), ?\omega \in \mathcal{POMSET}(L_i) : ?\omega \in \text{poss}(\bot_s \textbf{ after } \omega)$ and $?\omega \notin \text{poss}(\bot_i \textbf{ after } \omega)$

   If $?\omega \in \text{poss}(\bot_s \textbf{ after } \omega)$, either $(\omega \cdot ?\omega) \in \text{traces}(s)$ or $\exists ! \omega \in \mathcal{POMSET}(L_o) : (\omega \cdot !\omega \cdot ?\omega) \in \text{traces}(s)$ and by Hypothesis a) we have that exists a test case t where either $(\omega \cdot ?\omega) \in \text{traces}(t)$ or $(\omega \cdot !\omega \cdot ?\omega) \in \text{traces}(t)$. In both cases $?\omega \in \text{poss}(\bot_t \textbf{ after } \omega)$. Let t such that $\omega \in \text{traces}(t)$ and $?\omega \in \text{poss}(\bot_t \textbf{ after } \omega)$, we have

$$
\begin{array}{ll}
 & \exists \omega \in \text{traces}(t), ?\omega \in \mathcal{POMSET}(L_i) : ?\omega \in \text{poss}(\bot_t \textbf{ after } \omega) \text{ and} \\
 & ?\omega \notin \text{poss}(\bot_i \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 21 } *\} \\
 & \exists \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{I}}(i, t, \omega) \\
\text{implies} & \{* \text{ Definition 22 } *\} \\
 & i \textbf{ fails } T
\end{array}
$$

The interested reader may check that the test suite $\{t_4, t_5\}$ from Figure 13 is exhaustive and therefore complete.

6 Conclusion and Future Work

We have presented the formal framework for conformance testing over *concurrent* systems whose behavior is given in the form of labeled event structures. This continues and generalizes the work in [21, 20] where I/O automata were generalized such that I/O labeled partial orders were allowed to decorate transitions. The present work allows to drop entirely any assumptions about an underlying automaton formed by global states. Future technical studies include the question whether it is possible to drop assumption 4 and to handle the non-determinism thus introduced in the testing framework.

More generally, the framework given here will serve to accommodate I/O-enabled system models with concurrency - of the Petri net or network of automata type -, whose theory is part of future work.

Another important dimension to be explored is the distribution of observation and of testing. The **dioco** and associated relations studied by Hierons et al. [18,

32] allow to link the conformance of *local* observations to the *global* conformance of the SUT; but while there the underlying system model is a multi-port IOTS, we shall be studying multi-component, concurrent systems with local observation and distributed test suites to be developed.

References

1. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)
2. Hoare, T.: Communicating Sequential Processes. Prentice-Hall (1985)
3. ITU-TS: Recommendation Z.100: Specification and Description Language (2002)
4. Brinksma, E., Scollo, G., Steenbergen, C.: Lotos specifications, their implementations and their tests. In: Conformance testing methodologies and architectures for OSI protocols. IEEE Computer Society Press (1995) 468–479
5. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science **34** (1984) 83–133
6. Abramsky, S.: Observation equivalence as a testing equivalence. Theoretical Computer Science **53** (1987) 225–241
7. Brinksma, E.: A theory for the derivation of tests. In: Protocol Specification Testing and Verification VIII, North-Holland (1988) 63–74
8. Phillips, I.: Refusal testing. Theoretical Computer Science **50** (1987) 241–284
9. Langerak, R.: A testing theory for LOTOS using deadlock detection. In: Protocol Specification, Testing and Verification IX, North-Holland (1990) 87–98
10. Segala, R.: Quiescence, fairness, testing, and the notion of implementation. Information and Computation **138**(2) (1997) 194–210
11. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. Software - Concepts and Tools **17**(3) (1996) 103–120
12. De Nicola, R.: Extensional equivalences for transition systems. Acta Informatica **24**(2) (1987) 211–237
13. Heerink, A., Tretmans, G.: Refusal testing for classes of transition systems with inputs and outputs. In: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII). Volume 107 of IFIP Conference Proceedings;., London, Chapman & Hall (1997) 23–38
14. Lestiennes, G., Gaudel, M.C.: Test de systèmes réactifs non réceptifs. Journal Européen des Systèmes Automatisés **39**(1-2-3) (2005) 255–270
15. Faivre, A., Gaston, C., Le Gall, P., Touil, A.: Test purpose concretization through symbolic action refinement. In: Testing of Software and Communicating Systems. Volume 5047 of Lecture Notes in Computer Science., Springer (2008) 184–199
16. Jéron, T.: Symbolic model-based test selection. Electronic Notes in Theoretical Computer Science **240** (2009) 167–184
17. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design **34**(3) (2009) 238–304
18. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations for the distributed test architecture. In: Testing of Software and Communicating Systems. Volume 5047 of Lecture Notes in Computer Science., Springer (2008) 200–215
19. Ulrich, A., König, H.: Specification-based testing of concurrent systems. In: Formal Description Techniques for Distributed Systems and Communication Protocols. Volume 107 of IFIP Conference Proceedings., Chapman & Hall (1998) 7–22

20. von Bochmann, G., Haar, S., Jard, C., Jourdan, G.V.: Testing systems specified as partial order input/output automata. In: Testing of Software and Communicating Systems. Volume 5047 of Lecture Notes in Computer Science., Springer (2008) 169–183
21. Haar, S., Jard, C., Jourdan, G.V.: Testing input/output partial order automata. In: Testing of Software and Communicating Systems. Volume 4581 of Lecture Notes in Computer Science., Springer (2007) 171–185
22. Ponce de León, H., Haar, S., Longuet, D.: Conformance relations for labeled event structures. In: Tests and Proofs (TAP'12). Volume 7305 of Lecture Notes in Computer Science., Springer (2012) 83–98
23. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science **13** (1981) 85–108
24. Langerak, R., Brinksma, E.: A complete finite prefix for process algebra. In Halbwachs, N., Peled, D., eds.: Proceedings of CAV99. Volume 1633 of LNCS., Springer (1999)
25. Winskel, G.: Event structures. In: Advances in Petri Nets. Volume 255 of Lecture Notes in Computer Science. (1986) 325–392
26. Aceto, L., De Nicola, R., Fantechi, A.: Testing equivalences for event structures. In: Mathematical Models for the Semantics of Parallelism. Volume 280 of Lecture Notes in Computer Science. (1986) 1–20
27. Genc, S., Lafortune, S.: Distributed diagnosis of discrete-event systems using petri nets. In: ICATPN'03. (2003) 316–336
28. Jard, C., Jéron, T.: Tgv: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer **7** (2005) 297–315
29. Xu, Y., Stevens, K.S.: Automatic synthesis of computation interference constraints for relative timing verification
30. Heerink, A.W.: Ins and Outs in Refusal Testing. PhD thesis, Enschede (May 1998)
31. Tretmans, J.: Model based testing with labelled transition systems. In: Formal Methods and Testing. Volume 4949 of Lecture Notes in Computer Science. (2008) 1–38
32. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations and test generation for systems with distributed interfaces. Distrib. Comput. **25** (2012) 35–62 . DOI 10.1007/s00446-011-0149-1.