HAL
open science

Model Based Testing for Concurrent Systems with Labeled Event Structures

Hernán Ponce de León, Stefan Haar, Delphine Longuet

To cite this version:

Hernán Ponce de León, Stefan Haar, Delphine Longuet. Model Based Testing for Concurrent Systems with Labeled Event Structures. Journal of Software Testing, Verification and Reliability, 2014, Tests and Proofs in Model-based Testing, 24 (7), pp.558-590. 10.1002/stvr.1543 . hal-00914796v3

HAL Id: hal-00914796

https://inria.hal.science/hal-00914796v3

Submitted on 26 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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 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. We introduce the notions of strong and weak concurrency: strongly concurrent events must be concurrent in the implementation, while weakly concurrent ones may eventually be ordered. The **ioco** type conformance relations for sequential systems rely on the observation of sequences of actions and blockings, thus they are not capable of capturing and exploiting concurrency of non sequential behaviors. We propose an extension of **ioco** for labeled event structures, named **co-ioco**, allowing to deal with strong and weak concurrency. We extend the notions of test cases and test execution to labeled event structures, and give a test generation algorithm building a complete test suite 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 (SUT) 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 by Brinksma [7], where it was proposed to base the observations on the traces of the specification only, leading to a weaker conformance relation called **conf**. A further refinement concerns the inclusion of quiescent traces as a conformance relation [10]. Moreover, Tretmans proposed the **ioco** relation [11], 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 distinguished 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, 18], and multi-port finite state machines [19].

Model-based Testing of Concurrent Systems. Systems composed of several concurrent components 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 located on different components, 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 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.

Model-based testing of concurrent systems has been studied for a long time [20–22], however it is most of the time studied in the context of interleaving semantics, or trace semantics, which is known to suffer the state space explosion problem. 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.

Ulrich and König [23] propose a framework for testing concurrent systems specified by communicating labeled transition systems. They define a concurrency model called behavior machines that is an interleaving-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 [24, 25] 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 labeled 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, these models 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 follow here, continuing our previous work [26, **?**], proposes a formal framework for testing concurrent systems from true-concurrency models in which no synchronization on global states is required.

Weak vs. Strong Concurrency. As it is shown in Figure 1, concurrency can be implemented in two different ways. Boxes represent processes, letters are actions and a dependence between two actions is shown by an arrow. The three actions a, b and c are specified as concurrent in Spe (no dependence between them), actions a and c belonging to process P_1 while b belongs to process P_2.

In a distributed architecture, when two actions are specified as concurrent and belong to different processes, they should be implemented as concurrent, in different processes: this situation corresponds to the notion of *strong concurrency*, meaning that there should not be any kind of dependence between these actions. In Figure 1, actions a and b belong to different processes (i.e. they are strongly concurrent) in Spe and are implemented in different processes in both Impl_1 and Impl_2. In our previous work [27], concurrency is interpreted as strong concurrency, therefore the conformance relation forces concurrent actions to be implemented in different processes.

However, in an early stage of specification, concurrency between events may be used as underspecification. Actions belonging to the same component may be implemented in any order in the same process (as it is the case of a and c in Impl_1) or the specification may still be refined and this process implemented as several ones as it is the case of P_1 which is implemented as P_1' and P_1'' in Impl_2. We capture this kind of underspecification with *weak concurrency*. As it

is the case of local trace languages [28], in one situation two actions might be concurrent while in another situation they cannot be performed independently.

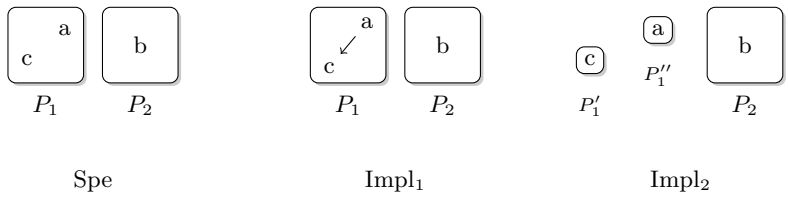


Fig. 1. A specification of a system and two possible implementations.

We illustrate the need to make these two notions of concurrency live in the same model by examples coming from the field of microcontroller design [29] and security protocols [30].

Example 1. Consider a *ParSeq controller* which manages two handshakes $A = (rek_a, ack_a)$ and $B = (rek_b, ack_b)$ according to a set of Boolean variables x_1, x_2, x_3 provided by the environment as shown in Figure 2. These variables are mutually exclusive (only one of them can be 1) and they decide how the handshakes are handled. If $x_1 = 1$, the handshake is initiated in parallel (concurrent events A **co** B), while any other possible valuation of the variables initiates the handshakes in sequence ($A < B$ if $x_2 = 1$ and $B < A$ if $x_3 = 1$). In this example events A and B would be specified as weakly concurrent, but their actual order would depend on the values of the variables rather than on an implementation choice. For more details about the controller see [29].

Example 2. When designing a security protocol, an important property, named unlinkability, is to hide the information about the source of a message. An attacker that can identify messages as coming from the same source can use this information and so 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 [30]. Causality captures linkability as two messages coming from the same user need to be causally dependent. However, concurrency interpreted as interleavings can not be used to model unlinkability because both possible interleavings relate the messages as if they were causaly dependent, therefore they reveal the identity of the user. This property needs to be modeled by strong concurrency.

Framework. 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.

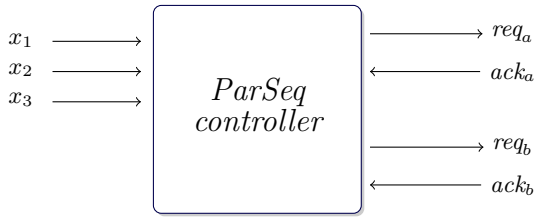


Fig. 2. ParSeq controller interface.

The underlying mathematical structure for the system semantics is given by *event structures* in the sense of Winskel et al [31]. 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 [31], and also as a natural semantics for process algebras (see e.g. the work of Langerak and Brinksma [32]). The state reached after some execution is represented by a *configuration* of the event structure, that is a conflict-free, history-closed set of events. The use of partial order semantics provides richer information and finer system comparisons than the interleaved view.

Our Contributions. We proposed in previous work [26] 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 under partial order semantics. We additionally dropped the input enabledness assumption and enlarged the conformance relation with the observation of refusals [27]. In this paper, we refine this conformance relation introducing the notions of strong and weak concurrency. Events specified as strongly concurrent must remain concurrent in a correct implementation while weakly concurrent events may be ordered. These two notions reflect the two usual interpretations of concurrency in a specification, that are true-concurrency semantics and interleaving semantics. These refinements lead to a new definition of the **co-ioco** conformance relation.

The contributions of this paper are twofold. First, we define the notions of strong and weak concurrency along with a new semantics for labeled event structures based on a notion of relaxed executions. Second, we define a whole framework for testing concurrent systems from labeled event structures. Besides the definition of a **co-ioco** conformance relation handling strong and weak concurrency, we define the notion of test case, we give sufficient properties for a test

suite to be sound (not rejecting correct systems) and exhaustive (not accepting incorrect systems), and we provide a test case generation algorithm that builds a complete (i.e. sound and exhaustive) test suite. The paper 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 the new notions of strong and weak concurrency, along with 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** relations of our previous papers [26, 27]. Section 5 develops the definitions of test cases and test suites, characterizing soundness, exhaustiveness and completeness of test suites, while Section 6 proposes an algorithm that builds a complete test suite, thus completing the contributions of the paper before Section 7 concludes.

2 Input/Output Labeled Event Structures

2.1 Syntax

We shall be using event structures following Winskel et al [31] to describe the dynamic behavior of a concurrent system. In this paper we will consider only prime event structures [33], 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 common practice 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 input/output labeled event structures over L by $\mathcal{IOLES}(L)$.

We assume that there exists a unique minimal element (w.r.t $\leq$), denoted and labeled by $\perp$, which is unobservable. The special label $\tau \notin L$ represents an unobservable (also called internal or silent) action. Given an event e, its past is

defined as $[e] \triangleq \{e' \in E \mid e' \leq e\}$. The sets of input, output and silent events are defined by $E^{\mathcal{I}} \triangleq \{e \in E \mid \lambda(e) \in L_{\mathcal{I}}\}, E^{\mathcal{O}} \triangleq \{e \in E \mid \lambda(e) \in L_{\mathcal{O}}\}$ and $E^{\tau} \triangleq \{e \in E \mid \lambda(e) = \tau\}$. When it is clear from the context, we will refer to an event by its label.

Two given 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. In this paper we split the **co** relation into two relations **sco** (strong concurrency) and **wco** (weak concurrency), such that **co** $=$ **sco** $\uplus$ **wco**.

Remark 1. The IOLES that we consider as the specification of the system is usually produced as the semantic unfolding from a language such as Petri nets, hence the specifier should provide the information about weak and strong concurrency as an annotation to the original specification.

The architecture of distributed systems allows to distinguish different components and there is a way to distinguish to which component each concurrent action belongs. Therefore, we make the following assumption.

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

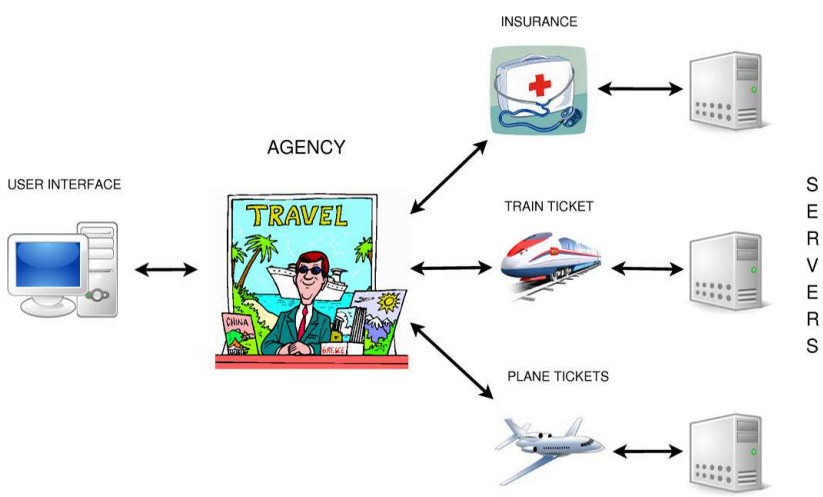


Fig. 3. A travel agency example.

Example 3. Figure 3 shows a schematic travel agency that sells services to customers on behalf of two suppliers, one selling both train and plane tickets and another one selling insurances. Its behavior can be formally specified by the IOLES s presented in Figure 4, 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 is chosen, its price is sent to the user ($!p_price$). If a train ticket is selected, the agency can internally decide (τ) what price to propose: a first class ($!t_price1$) or a second class one ($!t_price2$). The insurance choice is followed by its price ($!ins_price$) and some extra data that is sent to the user ($!ins_data$).

The data cannot be sent before the user logs 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 price can be produced ($!t_price1 \# !p_price$ and $!t_price2 \# !p_price$).

We consider strong concurrency only between the actions belonging to different suppliers: we do not want the selection of tickets to influence the prices of the insurance for example, therefore $\mathbf{sco} = \{?insurance, !ins_price, !ins_data\} \times \{?train, \tau, !t_price1, !t_price2, ?plane, !p_price\}$. All pairs of actions belonging to a single supplier (for tickets and insurance) are weakly concurrent with the $!us_data$ action. In addition the actions $!ins_price$ and $!ins_data$ are also weakly concurrent and we finally have $\mathbf{wco} = \{(!ins_data, !ins_price)\} \cup (\{!us_data\} \times \{?insurance, !ins_price, !ins_data, ?train, \tau, !t_price1, !t_price2, ?plane, !p_price\})$.

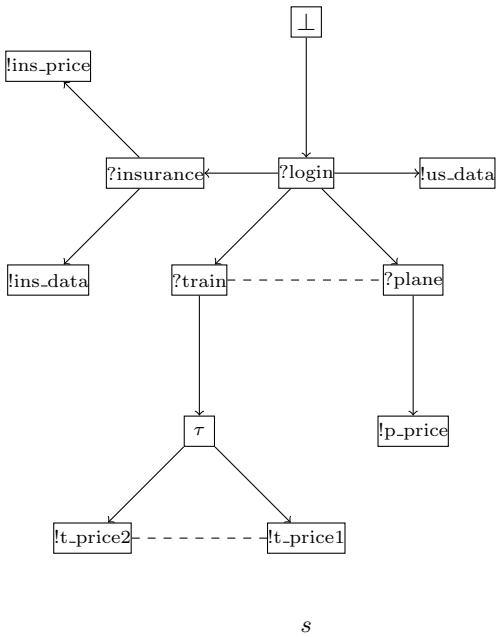


Fig. 4. Input/output labeled event structure of a travel agency.

Immediate Conflict. Most of the specification languages allow some way to model choice in the system. As conflict is inherited w.r.t causal dependency, a pair of events in conflict need not represent a choice between these events. We can see that $!t_price1$ and $!p_price$ are in conflict (by hierarchy), but any computation that continues by $!t_price1$, cannot continue by $!p_price$: the conflict was solved by the choice of $?train$ instead of $?plane$, and this makes $!p_price$ impossible. When the system makes a choice, we have a case of the *immediate conflict* relation in the following sense.

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*

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

Prefix of an IOLES. We define here the notion of prefix that allows to restrict the behavior of the system. As causality represents the events that should occur before a given event, the past of an event e that belongs to the prefix should also 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 $\forall e \in E' : [e] \subseteq E'$,*
- *$\leq' = \leq \cap (E' \times E')$,*
- *$\#' = \# \cap (E' \times E')$, and*
- *$\lambda' = \lambda_{|E'}$*

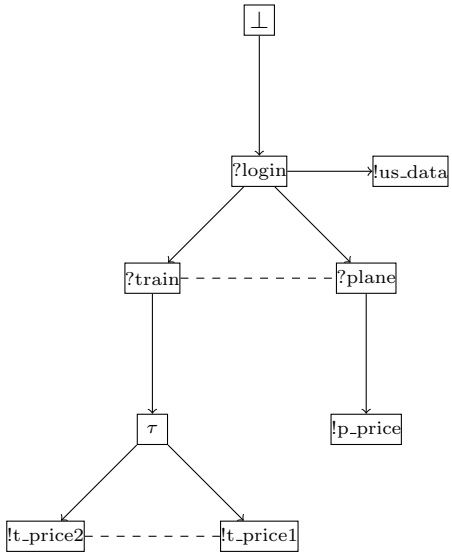
Example 4. In Figure 5, s_2 specifies the behavior of a travel agency that sells tickets, but not insurances, while the agency in s_3 only sells insurances. We can see that s_2 and s_3 are prefixes of s.

2.2 Semantics

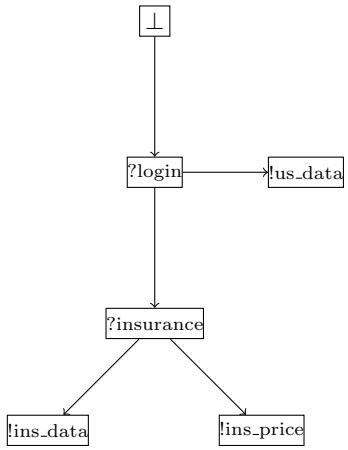
A computation state of an event structure is called a *configuration*; it is represented by the set of events that have occurred thus far in the computation. If an event is present in a configuration, then so are all the events on which this event causally depends (causal closure). Moreover, a configuration obviously does not contain conflicting events (conflict freedom). This is captured by the following standard definition [33].

Definition 4 (Configuration). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$. A configuration of $\mathcal{E}$ is a non-empty 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: $\forall e, e' \in C : \neg(e \# e')$.*



s_2



s_3

Fig. 5. Two prefixes of the travel agency.

A configuration C, equipped with the restriction of $\leq$, yields a partially ordered set, whose totally ordered extensions, or interleavings, describe possible sequential executions. Conversely, every sequential execution of the system is an interleaving of a unique configuration of the system; a configuration gives an equivalence class of possible interleavings. In this sense, configurations represent non-sequential executions.

Note that we define, for technical convenience, all configurations to be non-empty; the initial configuration of $\mathcal{E}$, containing only $\perp$ and denoted by $\perp_{\mathcal{E}}$, is contained in every configuration of $\mathcal{E}$. We denote the set of all the configurations of $\mathcal{E}$ by $\mathcal{C}(\mathcal{E})$.

Example 5. In the IOLES of Figure 4, after the user has logged in, the selections can be made while his information is sent to the server, i.e. $\{\perp, ?login, !us_data, ?insurance, ?train\} \in \mathcal{C}(s)$, but a train and a plane ticket cannot be selected in the same execution, i.e. $?plane, ?train \in C \Rightarrow C \notin \mathcal{C}(s)$. The configuration $\{\perp, ?login, !us_data, ?insurance, !ins_price, !ins_data, ?train$ is maximal (w.r.t $\subseteq$) as the remaining events are in conflict with the ones in the configuration.

LPOs and POMSETs. The definition of the notion of execution for an event structure is not straightforward since it relies on the chosen semantics for concurrency [34]. Here, we have two notions of concurrency, which impose partial orders and allow interleavings. We are interested in testing both kinds of concurrency and therefore we want to keep concurrency explicit in the executions. *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 this order as equivalent. Hence 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 isomorphim class of some LPO. Any such class is represented by one of its objects. We denote the class of all pomsets by* $\mathcal{POMSET}(L)$.

When it is clear from the context, we will use "$\cdot$" to express causality between pomsets and "**co**" to represent the pomset whose elements are unordered. The pomset μ_4 of Figure 6 can be represented by $\perp \cdot ?login \cdot !us_data \cdot ?insurance \cdot (!ins_price \ \mathbf{co} \ !ins_data)$.

As weak concurrency allows to order events, the selection of a ticket in the travel agency can be done after sending the data and therefore μ_1 and μ_2 from Figure 6 should be treated as equal (it is also the case with μ_3, μ_4, μ_5 and μ_6). For this reason, an execution of this specification has to preserve the partial order semantics of the IOLES, up to adding order between weakly concurrent events (while strongly concurrent events must remain concurrent).

Definition 8 (Relaxed 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*

- $\leq_{\mu_1} \subseteq \ \leq_{\mu_2}$
- $\boldsymbol{sco}_1 = \boldsymbol{sco}_2$

In other words, $\mu_1 \sqsubseteq \mu_2$ if strong concurrency is preserved; while weakly concurrent events from μ_1 may be ordered by $\leq_{\mu_2}$.

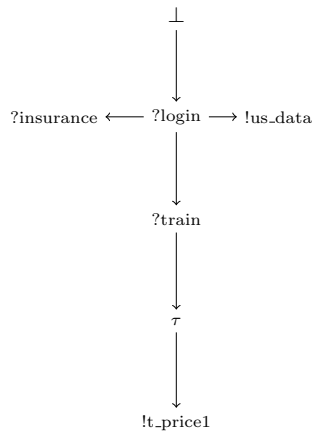
Example 6. We see in Figure 6 that μ_2 adds some ordering between weakly concurrent events $!us_data$ and $?train$ from μ_1, but strong concurrency and causality are preserved, and therefore $\mu_1 \sqsubseteq \mu_2$. The same order is added from μ_3 in μ_4, and μ_5 adds an ordering between the weakly concurrent events $!ins_price$ and $!ins_data$. Since no other relations are changed, $\mu_3 \sqsubseteq \mu_4$ and $\mu_4 \sqsubseteq \mu_5$. As $\sqsubseteq$ is transitive, we have $\mu_3 \sqsubseteq \mu_5$. In μ_6, $!us_data$ is preceded by $?insurance$ and then $\mu_3 \sqsubseteq \mu_6$.

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*.

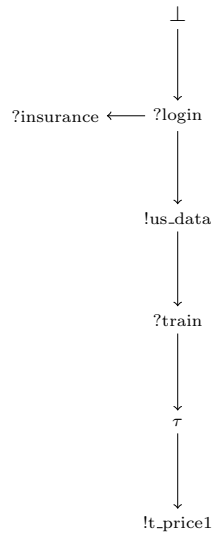
Definition 9 (Relaxed execution). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L), \mu, \mu'' \in \mathcal{POMSET}(L)$ *and* $C, C', C'' \in \mathcal{C}(\mathcal{E})$*, we define*

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

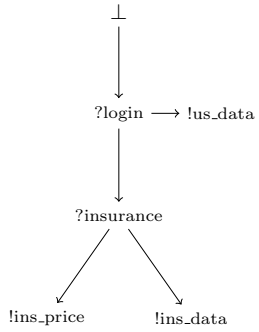
We say that μ *is a relaxed execution of* C *if* $C \xrightarrow{\mu}$.



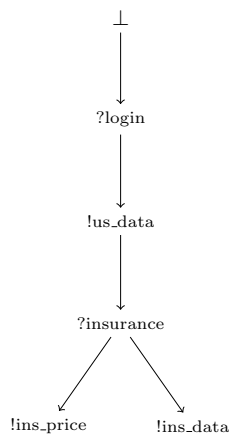
μ_1



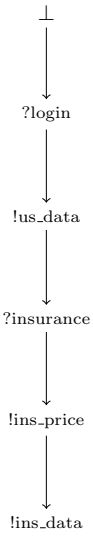
μ_2



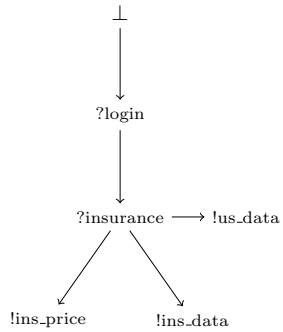
μ_3



μ_4



μ_5



μ_6

From the definition above we get that whenever $C \xrightarrow{\mu}$ and $\mu \sqsubseteq \mu'$, we have $C \xrightarrow{\mu'}$.

Example 7. In Figure 6 we can see that μ_1 and μ_3 respect the structure of s of Figure 4, and therefore both are relaxed executions of it ($\bot_s \xrightarrow{\mu_1}$ and $\bot_s \xrightarrow{\mu_3}$). However, as seen earlier, $\mu_1 \sqsubseteq \mu_2, \mu_3 \sqsubseteq \mu_4, \mu_3 \sqsubseteq \mu_5$ and $\mu_3 \sqsubseteq \mu_6$; therefore μ_2, μ_4, μ_5 and μ_6 are also relaxed executions of s ($\bot_s \xrightarrow{\mu_2}, \bot_s \xrightarrow{\mu_4}, \bot_s \xrightarrow{\mu_5}$ and $\bot_s \xrightarrow{\mu_6}$). In the case of μ_6, we see that our semantics allows an execution where an output ($!us_data$) depends on an extra input ($?insurance$). However, we will see later that our conformance relation prevents an output from depending on an extra input, even if these events are specified as a weakly concurrent pair. We can conclude that the same structure can have several relaxed executions where weakly concurrent events are ordered.

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], Heerink and Tretmans [13] and Lestiennes and Gaudel [14] propose conformance relations that in addition considers the actions that the system *refuses*. 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 cannot evolve without the interaction of the environment, usually called *quiescence* following Segala [10]. In this section, we define these three notions in the context of labeled event structures before presenting the **co-ioco** conformance relation in Section 4.

3.1 Traces

The labels in L represent the observable actions of a system; they model the interactions of the system with its environment while internal actions are denoted by the special label $\tau \notin L$. The observable behavior can be captured by abstracting the internal actions from the executions of the system (which are pomset in our setting).

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 executions.

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*

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

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

In the **ioco** theory, ? and ! are used to denote input and output actions respectively. We extend this notation and denote by $?\omega$ and $!\omega$ observations composed only of 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 trace is similar to the one of Ulrich and König [23] 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 [35].

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*

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

Example 8. Consider the pomsets of Figures 6 and 7. Clearly, $abs(\mu_1) = \omega_1$, and we saw in Example 7 that $\bot_s \xrightarrow{\mu_1}$; therefore, $\bot_s \overset{\omega_1}{\Longrightarrow}$. The same is true for μ_3, μ_6 and ω_3, ω_6. Thus $\omega_1, \omega_3, \omega_6 \in \text{traces}(s)$.

The configuration reached in s after the observations ω_3 and ω_6 is the same, i.e. $(\bot_s \textbf{ after } \omega_3) = (\bot_s \textbf{ after } \omega_6) = \{\{\bot, ?login, !us_data, ?insurance, !ins_price, !ins_data\}\}$. This example shows that even if the way of observing executions are different (due to weak concurrency), they all come from the same structure and lead to the same configuration.

Our definition of **after** is general enough to handle nondeterminism in the computation, however in this paper, for technical convenience, we will only consider specifications where exactly one configuration can be reached after some observation. Such systems are called *deterministic*.

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

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

When the set of reachable configurations is a singleton $\{C\}$ we will simply denote it by C.

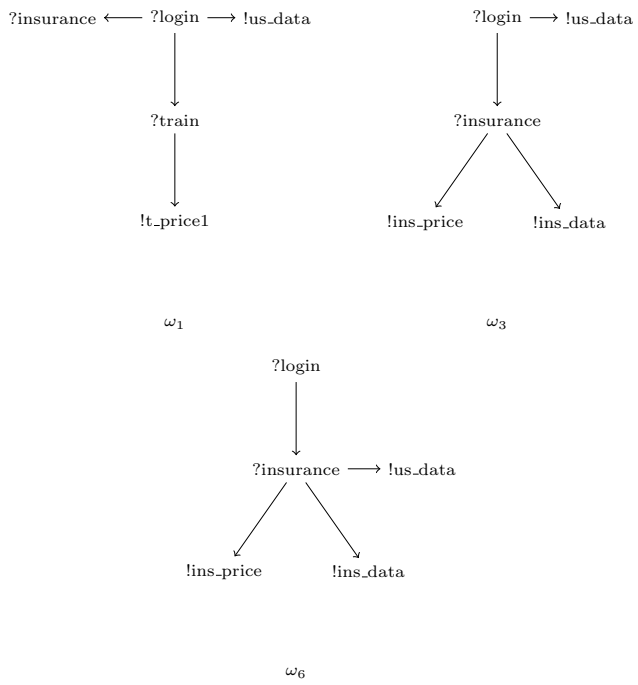


Fig. 7. Traces of the travel agency

Example 9. We can see from s_4 in Figure 8 that the observation makes no 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 nondeterministic.

In the **ioco** framework, the specification is determinized before the construction of test cases. As the IOLES that we consider are usually produced as the semantic unfolding of the specification, we assume that determinization is done directly on the specification and the resulting IOLES is deterministic.

Assumption 2 *The specification of the system is deterministic, i.e.* $\forall \omega \in traces(s)$: $(\perp_s \textbf{\textit{after}}\ \omega)$ *is a singleton.*

3.2 Quiescence and Produced Outputs

Since the testing activity depends on the interaction between the tester and the system, such an interaction becomes impossible if we allow the system to have infinitely many occurrences of silent or output actions without input ones. Therefore we make the following assumption.

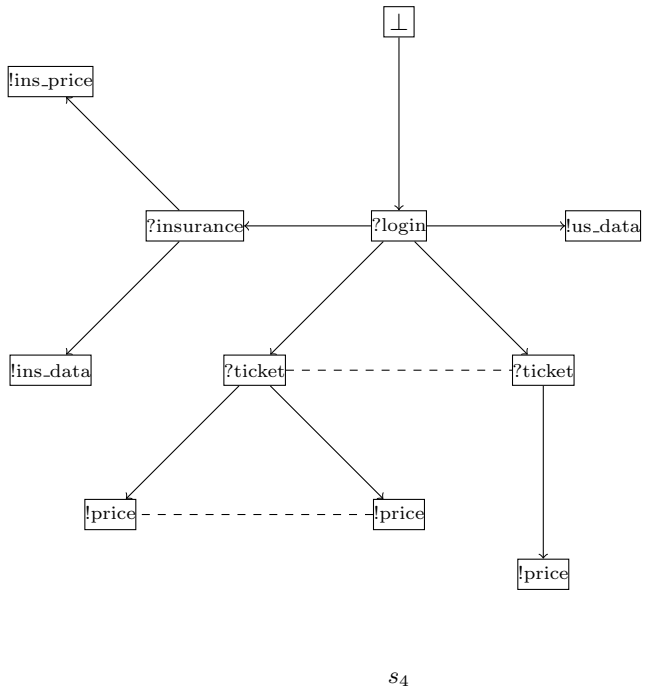


Fig. 8. A nondeterministic IOLES.

Assumption 3 *We will only consider systems that cannot diverge by infinitely many occurrences of silent or output actions, i.e.* $\forall C \in \mathcal{C}(\mathcal{E})$: *if* $C \cap (E^{\mathcal{O}} \cup E^{\tau})$ *is infinite then so is* $C \cap E^{\mathcal{I}}$.

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

With reactive systems, we need to differentiate configurations where the system can still produce some outputs, and those where the system cannot 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 implemented by timers. Jard and Jéron [36] present three different kinds of quiescence: *output quiescence*: the system is waiting for an input from the environment, *deadlock*: the system cannot evolve, and *livelock*: the system diverges by an infinite sequence of silent actions. Both output quiescence and deadlock are captured by the definition below, while livelock is not possible by Assumption 3.

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 define the δ action not by loops, but rather semantically: the δ action does not represent an event, and thus no new configuration is reached after observing it.

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.* C *is quiescent iff* $C \overset{\delta}{\Longrightarrow}$.

Example 10. In the travel agency example, the configuration reached after logging in is not quiescent as there is an execution where the user's data can be sent, i.e $(\bot_s \textbf{ after } ?login) = \{\bot, ?login\}$ and $\{\bot, ?login\} \overset{!us_data}{\Longrightarrow}$, but the configuration reached after sending the user's data is quiescent because only input actions are enabled in all possible executions, i.e. $(\bot_s \textbf{ after } (?login \cdot !us_data)) = \{\bot, ?login, !us_data\}$ and for every ω such that $\{\bot, ?login, !us_data\} \overset{\omega}{\Longrightarrow}$, we have $E^{\mathcal{I}}_{\omega} \neq \emptyset$.

In the LTS framework, the *produced outputs* of the systems are single elements of the alphabet of outputs rather than sequences of them [11]. Consider a system s that produced $!a$ followed by $!b$ after σ, then $\text{out}(s \textbf{ after } \sigma) = \{!a\}$ and $\text{out}(s \textbf{ after } (\sigma \cdot !a)) = \{!b\}$ rather than $\text{out}(s \textbf{ after } \sigma) = \{!a \cdot !b\}$. A first extension proposed by the authors [26] considers that the outputs produced by the system in response to stimuli could be elementary actions as well as sets of concurrent actions. However, here 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. In fact, suppose one has two concurrent outputs out_1 and out_2 depending on input in_1 and another input

in_2 depending on both outputs. Clearly, an implementation that accepts in_2 before out_2 should not be considered as correct, but if in_2 is sent too early to the system, we may not know if the occurrence of out_2 depends or not on in_2. For this reason, Definition 15 defines the expected outputs from a configuration as the pomset of outputs leading to a quiescent configuration. Such a configuration always exists, and is finite by Assumption 3.

However, conformance of output pomsets is not always captured by isomorphism. Consider again the example presented in the paragraph above and consider out_1 and out_2 as weakly concurrent. After in_1 the system produces outputs out_1 and out_2 which can be observed concurrently or in any order (due to the relaxed executions). We want to compare the produced outputs of the implementation with those of the specification, but as we allow the implementation to order these outputs, these set can not be directly compared by set inclusion. Any produced output in the implementation should refine some produced output in the specification (see next section). Here, both $out_1 \cdot out_2$ and $out_2 \cdot out_1$ can be inferred from out_1 **wco** out_2. We only consider out_1 **wco** out_2, which is the "most abstract" pomset representing both orders, i.e. the minimal pomset w.r.t. $\sqsubseteq$, as it is sufficient to compare outputs w.r.t refinement.

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

$$out(C) \triangleq \min_{\sqsubseteq}\{!\omega \in \mathcal{POMSET}(L_o) \mid C \xRightarrow{!\omega} C' \ \wedge \ C' \xRightarrow{\delta}\} \cup \ \{\delta \mid C \xRightarrow{\delta}\}$$

Example 11. The only output produced by the system of Figure 4 after logging in is the user's data, i.e. out($\bot_s$ **after** $?login$) = $\{!us_data\}$ and after this output, a quiescent configuration is reached, then out($\bot_s$ **after** $(?login \cdot !us_data)$) = $\{\delta\}$. If a train ticket is chosen, different prices may be produced, i.e. out($\bot_s$ **after** $(?login \cdot !us_data \cdot ?train)$) = $\{!t_price1, !t_price2\}$. The outputs after selecting the insurance are weakly concurrent and can be observed in different ways (concurrently or in any order), however we only consider the $\sqsubseteq$-minimal outputs, and therefore out($\bot_s$ **after** $(?login \cdot !us_data \cdot ?insurance)$) = $\{!ins_price$ **co** $!ins_data\}$.

3.3 Refusals

The **ioco** theory assumes the input enabledness of the implementation, i.e. in any state of the implementation, every input action is enabled. This assumption is made to avoid computation interference [37] in the parallel composition between the implementation and the test cases. However, as explained by Heerink [38] and Lestiennes and Gaudel [14], even if many realistic systems can be modeled with such an assumption, there remains a significant portion of realistic systems that cannot. 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 not very realistic situation in a distributed environment.

In order to overcome these difficulties, Heerink [38] distributes the points of control and observation, and the input enabledness 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 a special ξ-action (as quiescence is observable by a δ action). Lestiennes and Gaudel [14] 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 the one of Lestiennes and Gaudel; 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). This implies that we assume that there exists a way to observe the refusal of an input by the implementation during testing. This assumption is quite natural, for instance in the case of the cash dispenser which cannot accept more than one card. One can consider that the system under test would display an error message or a warning in case it cannot handle an input the test sends.

In an observation, an input action may be preceded by an output that was specified to be weakly concurrent as it is the case with $!us_data$ and $?train$ in ω_2. Therefore $?train$ should still be considered as possible even if the $!us_data$ output has not been produced yet. This is similar in remote testing [39] where communication between test cases and the SUT is asynchronous and a new input can be sent even if an output that precedes it was not still produced.

The possible inputs of a configuration are those that are enabled or will be enabled after producing some outputs. As in the case of produced outputs, we consider the set of $\sqsubseteq$-minimal inputs.

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

$$poss(C) \triangleq \min_{\sqsubseteq}\{?\omega \in \mathcal{POMSET}(L_i) \mid C \overset{?\omega}{\Longrightarrow} \vee \exists !\omega \in \mathcal{POMSET}(L_o) : C \overset{!\omega}{\Longrightarrow} C' \wedge C' \overset{?\omega}{\Longrightarrow}\}$$

Example 12. Consider the IOLES s of Figure 4. The first possible input is the logging in, followed by the selections of tickets and insurance. These selections are possible either alone or concurrently, i.e. $\text{poss}(\bot_{i_1}) = \{?login, ?login \cdot ?insurance, ?login \cdot ?train, ?login \cdot ?plan$ $(?insurance\ \mathbf{co}\ ?train), ?login \cdot (?insurance\ \mathbf{co}\ ?plane)\}$.

In order to allow the observation of the possible inputs of the system under test, a configuration where inputs are possible should not alternatively allow 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 would 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 4 *We will only consider IOLES such that there is no immediate conflict between input and output events, i.e.* $\forall e \in E^{\mathcal{I}}, e' \in E^{\mathcal{O}} : \neg(e \overline{\#} e')$.

Note that a similar assumption was also made by Gaudel et al [14], where such specifications are called IO-exclusive. Under assumption 4, no enabled input can be *disabled* by the production of outputs: to disable an input, some *conflicting input* must be made.

Proposition 1. *Let* $\mathcal{E} \in \mathcal{IOLES}(L)$ *such that* $\mathcal{E}$ *satisfies Assumption 4. Let* $C, C' \in \mathcal{C}(\mathcal{E})$ *such that there exists* $!\omega \in \mathcal{POMSET}(L_o)$, $C \overset{!\omega}{\Longrightarrow} C'$ *and* C' *is quiescent. Then any possible input in* C *is a possible input in* C'*, i.e.* $poss(C) = poss(C')$.

Proof. Let us assume that $?\omega \in \text{poss}(C)$ for a non quiescent configuration C. We have then that either $C \overset{?\omega}{\Longrightarrow}$, or we can reach from C a quiescent configuration C' such that $C' \overset{?\omega}{\Longrightarrow}$. The result is immediate for the second case. 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 possible observations $?\omega$ and $!\omega$ at the same configuration C, and hence its events must be in immediate conflict or concurrent. By the assumption, 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 that is based on the observation notions of traces, refusals and quiescence.

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 [34] propose several testing equivalences depending in the chosen semantics for event structures. The authors [26] propose two extensions for the **ioco** conformance relation proposed by Tretmans [40], one for the interleaving semantics and another for the partial order one. The input enabledness assumption can be dropped and the conformance relation enlarged in order to observe refusals [27]. Actions specified as concurrent must occur independently (on different processes) in any conformant implementation. Here, since we refine the semantics of IOLES with strong and weak concurrency, we need to refine the conformance relation **co-ioco** in order to take these two interpretations of concurrency into account.

Our 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:

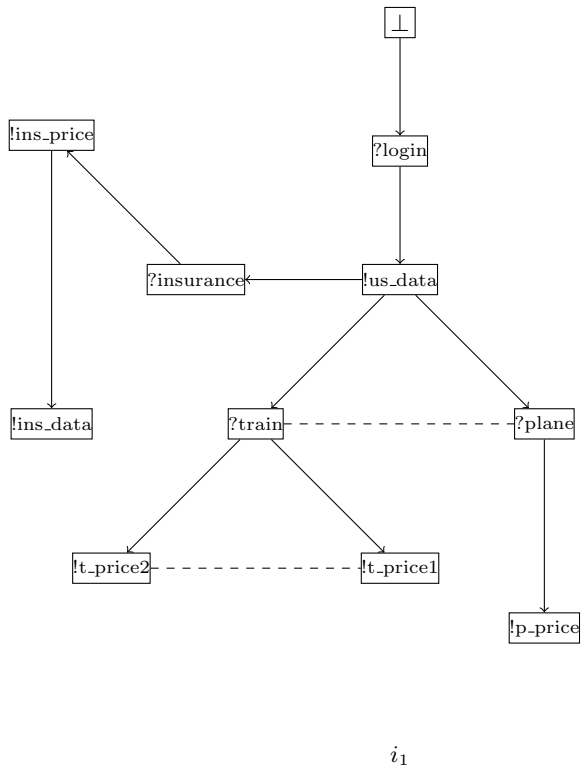


Fig. 9. A correct implementation w.r.t **co-ioco** of the travel agency of Figure 4.

1. any output produced by the implementation should be produced by the specification;
2. if a quiescent configuration is reached in the implementation, 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;
4. strongly concurrent events are implemented concurrently, while weakly concurrent events may be ordered.

Before the definition of the conformance relation itself, we need a few more technical definitions in order to be able to compare the inputs and outputs of the system under test to those of its specification.

Concurrent Completeness. As two inputs may be weakly concurrent in the specification, we want to accept an implementation where they are only implemented in one order. Suppose there exists two weakly concurrent inputs in_1, in_2 such that $\perp_s \overset{in_1 \ \mathbf{co} \ in_2}{\Longrightarrow}$, then $\text{poss}(\perp_s) = \{in_1, in_2, in_1 \ \mathbf{co} \ in_2\}$. Now consider an implementation that orders them, e.g. $\text{poss}(\perp_i) = \{in_1, in_1 \cdot in_2\}$. We cannot compare

the possible inputs of both systems w.r.t set inclusion because we want inputs to be implemented in *at least* one of the allowed orders.

We expect any pomset of possible inputs in the specification to be implemented either as such or as one of its refinements. However, this is not enough. In the example presented in the paragraph above there is no possible input of the implementation (in its initial configuration) that refines in_2. In addition, our definition of possible inputs accepts partial orders with some causality as in the case of s where we have $?login \cdot (?insurance \ \mathbf{co} \ ?train) \in \text{poss}(\bot_s)$. However, we may add some order for a weakly concurrent output (as is the case of $!us_data$ in i_1) and therefore we cannot find a possible input in the implementation that refines one of the specification. For this reason, we restrict to *concurrent complete sets* of inputs.

Definition 17 (Concurrent Complete Set). *Let $\omega \in \mathcal{POMSET}(L)$ and $C \in \mathcal{C}(\mathcal{E})$, we say that ω is a concurrent complete set in C iff any other execution from C (without causality) does not contain events that are concurrent to those of ω*

$$cc(\omega, C) \Leftrightarrow \omega = \max_{\subseteq}\{\omega \mid C \overset{\omega}{\Longrightarrow} \wedge \ \leq_\omega = \emptyset\}$$

Example 13. Consider i_1 from Figure 9. There, $?insurance$ is possible after logging in and sending the data, i.e. $(\bot_{i_1} \ \mathbf{after} \ (?login \cdot !us_data)) \overset{?insurance}{\Longrightarrow}$. However this input is not a concurrent complete set as it can be "extended" by concurrent events, i.e. $(\bot_{i_1} \ \mathbf{after} \ (?login \cdot !us_data)) \overset{?insurance \ \mathbf{co} \ ?train}{\Longrightarrow}$.

Now, possible inputs are checked in several steps: we first find a refinement for $?login$ from the initial configuration, and later a refinement for $?insurance \ \mathbf{co} \ ?train$ from $\{?login\}$.

As explained above, the possible inputs of the specification cannot be directly compared with those of the implementation. We want any concurrent complete input of the specification without causality to be implemented by one of its refinements (as $\sqsubseteq$ is reflexive, the input can be implemented as it is specified).

Definition 18 (Input refinement). *Let $C, C' \in \mathcal{C}(\mathcal{E})$ we define*

$$poss(C) \gg^+ poss(C') \Leftrightarrow \forall ?\omega \in poss(C) : (cc(?\omega, C) \Rightarrow \exists ?\omega' \in poss(C') : \omega \sqsubseteq \omega')$$

Analogously, outputs cannot be compared directly by set inclusion; we need every output produced by the implementation to refine some output of the specification.

Definition 19 (Output abstraction). *Let $C, C' \in \mathcal{C}(\mathcal{E})$ we define*

$$out(C) \gg^- out(C') \Leftrightarrow \forall x \in out(C) : \exists x' \in out(C') : x' \sqsubseteq x$$

Notice that δ only refines itself, therefore if $\delta \in \text{out}(C)$ and $\text{out}(C) \gg^- \text{out}(C')$ then $\delta \in \text{out}(C')$.

The ***co-ioco*** *Conformance Relation.* Now requirements 1, 2, 3 and 4 can be formalized by the following conformance relation.

Definition 20 (co-ioco). *Let* $i, s \in \mathcal{IOLES}(L)$*, then*

$$\begin{aligned} i \ \textbf{\textit{co-ioco}}\ s \Leftrightarrow \forall \omega \in traces(s) : \\ poss(\bot_s \ \textbf{\textit{after}}\ \omega) \gg^{+} poss(\bot_i \ \textbf{\textit{after}}\ \omega) \\ out(\bot_i \ \textbf{\textit{after}}\ \omega) \gg^{-} out(\bot_s \ \textbf{\textit{after}}\ \omega) \end{aligned}$$

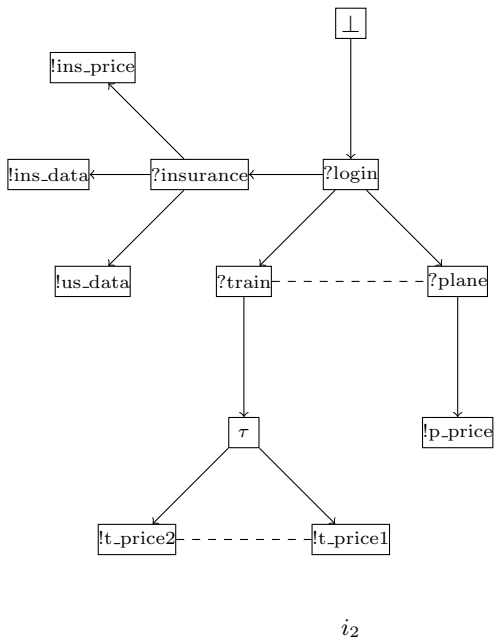


Fig. 10. Output depending on extra input.

Example 14 (Order of Weakly Concurrent Events). The implementation i_1 of the travel agency proposed in Figure 9 orders some weakly concurrent events. The outputs $!ins_price$ and $!ins_data$ are implemented sequentially instead of concurrently, but the output produced ($!ins_price \cdot !ins_data$) refines an output produced by the specification ($!ins_price$ **co** $!ins_data$). Some order is also added between inputs $?insurance, ?train, ?plane$ and output $!us_data$. However, these inputs depend on the output, and therefore the produced outputs in the implementation are those specified. Even if some order is added, by Proposition 1, every possible input of the specification is implemented. We can conclude that i_1 **co-ioco** s.

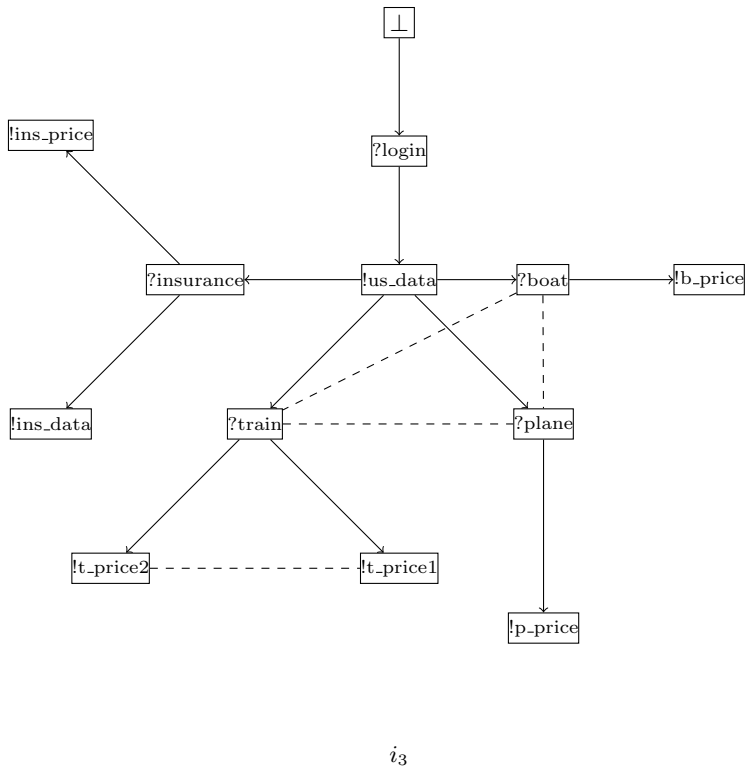


Fig. 11. Extra conflicting inputs.

Events $?insurance$ and $!us_data$ are ordered in the opposite way in implementation i_2 of Figure 10 ($?insurance \leq !us_data$). A quiescent configuration is reached after logging in, i.e. $\text{out}(\bot_{i_2}$ **after** $?login) = \{\delta\}$ while $\text{out}(\bot_s$ **after** $?login) = \{!us_data\}$ and then $\neg(i_2$ **co-ioco** $s)$. We can conclude that whenever an input and an output are weakly concurrent, then if ordering is added in a conformant implementation, the output should precede the input.

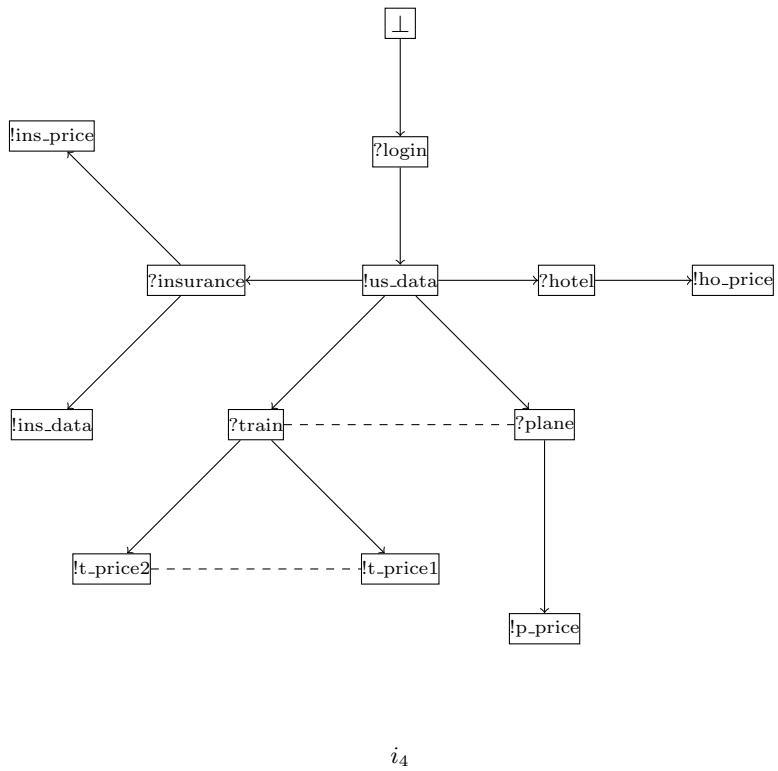


Fig. 12. Extra concurrent inputs.

Example 15 (Extra Inputs). The behaviors of the implementation and the specification are compared after some observations are made. These observations are taken from the specification (they are traces of s in Figure 4), therefore, there is no restriction for the implementation about how to react to unspecified inputs. Figure 11 shows a possible implementation i_3 that also allows the user to choose a boat ticket. Even if this implementation may produce an extra output ($!b_price$), this output is only produced after the boat ticket has been chosen, but the behavior of the system after choosing a boat ticket is not specified. Finally, we have i_3 **co-ioco** s. Figure 12 presents implementation i_4 that allows the user to

concurrently chose for a hotel. We consider concurrent complete possible inputs only in the specification; thus the $?hotel$ action and its corresponding output are never tested. As every concurrent complete possible input of the specification is refined by one of the implementations, we can conclude that i_4 **co-ioco** s.

Example 16 (Refused Inputs). The conformance relation considers the input actions that the implementation may refuse. Figure 13 presents two possible implementations i_5 and i_6 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 part of any possible input in i_5, i.e. $\text{poss}(\bot_{i_5} \textbf{ after } ?login) = \{?train, ?plane\}$ and finally $\neg(i_5 \textbf{ co-ioco } s)$. The $?train$ action is neither part of a possible input in i_6, i.e. $\text{poss}(\bot_{i_6} \textbf{ after } ?login) = \{?insurance, ?plane, ?insurance \textbf{ co } ?plane\}$ and then $\neg(i_6 \textbf{ co-ioco } s)$.

Example 17 (Extra/incomplete Outputs). The second condition of the conformance relation establishes that all the outputs produced by the implementation should be specified. Consider the implementation i_7 presented in Figure 14: after choosing the plane ticket, the implementation can produce an output with the ticket price, or an error message due to the fact that there are no tickets available, i.e. $\text{out}(\bot_{i_7} \textbf{ after } (?login \cdot !us_data \cdot ?plane)) = \{!p_price, !p_full\}$, but $!p_full$ is not a possible output in the specification, i.e. $!p_full \notin \text{out}(\bot_s \textbf{ after } (?login \cdot !us_data \cdot ?plane)) = \{!p_price\}$, therefore $\neg(i_7 \textbf{ co-ioco } s)$. The conformance relation only considers "complete" outputs (those that lead to a quiescent configuration), while incomplete outputs lead to non conformance. Implementation i_7 also shows an example of this: after choosing the insurance, only its price is produced, i.e. $\text{out}(\bot_{i_7} \textbf{ after } (?login \cdot !us_data \cdot ?insurance)) = \{!ins_price\}$. This output does not refine any produced output of the specification, as some insurance data should also be produced (either concurrently or in some order), i.e. $\text{out}(\bot_s \textbf{ after } (?login \cdot !us_data \cdot ?insurance)) = \{!ins_price \textbf{ co } !ins_data\}$, and again $\neg(i_7 \textbf{ co-ioco } s)$.

Example 18 (Extra Quiescence). The second condition of the conformance relation stipulates that *absence* of outputs can only occur when it is specified. Figure 15 shows an implementation i_8 that does not send the user's data after logging in, thus a quiescent configuration is reached after logging in, i.e. $\text{out}(\bot_{i_8} \textbf{ after } ?login) = \{\delta\}$, but this quiescence is not specified, i.e. $\delta \notin \text{out}(\bot_s \textbf{ after } ?login) = \{!us_data\}$, and $\neg(i_8 \textbf{ co-ioco } s)$.

Comparing the Conformance Relations. We present now a comparison between the previous conformance relations [26, 27] and the one presented in this paper. The first **co-ioco** conformance relation [26] allows two different semantics. Under interleaving semantics this relation boils down to **ioco** while partial order

semantics allows to distinguish true concurrency from interleavings. The second notion of conformance [27] only considers partial order semantics: events specified as concurrent should be implemented as such. In addition the input enabledness assumption of the implementation is dropped and we allow to test for refusals. Finally, as explained above, under the definition of **co-ioco** we give in this paper, implementations where events specified as weakly concurrent are ordered are considered as correct.

When every pair of concurrent event is specified as strongly concurrent (there are not weakly concurrent events), the **co-ioco** relation presented in this paper boils down to the second conformance relation [27]. In addition, if we assume that the implementation is input enabled, then the first [26] and second conformance relation [27] (with partial order semantics) are equivalent. Finally, when there is no concurrency at all (the system is sequential), the first relation [26] boil down to **ioco**. These results are summarized in the following table.

Assumptions	**Results**
co = **sco**	**co-ioco** = [27]
co = **sco** $\wedge$ input enabledness of the SUT	**co-ioco** = [27] = [26]
co = **sco** $\wedge$ input enabledness of the SUT $\wedge$ **co** = $\emptyset$	**co-ioco** = [27] = [26] = **ioco**

5 A Testing Framework for 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 notions of test cases and test suites, as well as their interaction with the implementations and we give sufficient conditions for detecting all and only incorrect implementations.

In order to formally reason about implementations, we make the testing assumption that the implementation under test can be modeled by an IOLES.

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 observed, but not controlled by the tester; however, the tester does control the input ones. It follows that there should be no choices between them, i.e. the next (set of concurrent) input(s) to be proposed should be unique, therefore no *immediate conflict between inputs* should exist in a test case.

[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.

This property is not enough to avoid all 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 (nondeterministic) choice for the tester about the next input to propose even if those inputs are not in immediate conflict. We require thus determinism: the reached configuration after some observation should be unique.

Finally, we require the experiment to finish, therefore the test case 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 21 (Test Case / Test Suite). *A test case is a input/output labeled event structure* $t = (E_t, \leq_t, \#_t, \lambda_t)$ *such that*

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

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

Example 19. Figure 16 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 differently. As in the case of quiescence, we do not define verdicts syntactically, 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 assumed that the implementation accepts any input the tester can propose (input enabledness 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 **ioco** framework. Such deadlocks are produced in the following situations:

1. the implementation proposes an output or a δ action that the test case cannot accept,
2. the test case proposes an input that the implementation cannot 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 obtaining 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 the implementation produces for three reasons. First, if after such an observation the test case cannot accept that output. Second, the test case can accept such output, but this is not the maximal output it can accept (the reached configuration is not quiescent). Finally the test execution blocks if the implementation reaches a quiescent configuration and the test case does not.

Such situations can be simplified to the observation of an element in the set of outputs produced by the implementation that does not refine any output of the test case, i.e. $\exists x \in \text{out}(\bot_i \textbf{ after } \omega) : \forall x' \in \text{out}(\bot_t \textbf{ after } \omega) : \ x' \not\sqsubseteq \ x$ where $x \in \mathcal{POMSET}(L_o) \cup \{\delta\}$.

Definition 22 (Blocking because of an output). *Let $i, t \in \mathcal{IOLES}(L)$ and $\omega \in \mathcal{POMSET}(L)$, we have*

$$\textbf{\textit{blocks}}_{\mathcal{O}}(i, t, \omega) \Leftrightarrow \textit{out}(\bot_i \textbf{ \textit{after} } \omega) \not\gg^{-} \textit{out}(\bot_t \textbf{ \textit{after} } \omega)$$

Example 20. Consider the implementation i_7, the test case t_4 presented in Figure 17, 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 $!p_full$ action (which leads to a quiescent configuration) and the test case is not able to accept it, i.e. $!p_full \in \text{out}(\bot_{i_7} \textbf{ after } \omega'_1)$, but $\forall x \in \text{out}(\bot_{t_4} \textbf{ after } \omega'_1) : x \not\sqsubseteq !p_full$, and finally $\textbf{blocks}_{\mathcal{O}}(i_7, t_4, \omega'_1)$. 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. However, the reached configuration is not quiescent because $!ins_data$ is still enabled, i.e. $!ins_price \in \text{out}(\bot_{i_7} \textbf{ after } \omega'_2), \text{out}(\bot_{t_4} \textbf{ after } \omega'_2) = \{!ins_price \textbf{ co } !ins_data\}$ and $!ins_price \not\sqsubseteq !ins_price \textbf{ co } !ins_data$. Finally $\textbf{blocks}_{\mathcal{O}}(i_7, t_4, \omega'_2)$.

Blocking because of an output can also be caused by extra quiescence. Consider the implementation i_2 and the test case t_4. We have that the test execution blocks after $?login$ because the implementation reaches a quiescent configuration, i.e. $\delta \in \text{out}(\bot_{i_2} \textbf{ after } ?login)$, but δ is not observable in the test case, i.e. $\delta \notin \text{out}(\bot_{t_4} \textbf{ after } ?login)$, and $\textbf{blocks}_{\mathcal{O}}(i_2, t_4, ?login)$. The same holds for i_8 and we have $\textbf{blocks}_{\mathcal{O}}(i_8, t_4, ?login)$.

The second blocking situation occurs when the test case proposes a concurrent complete set of inputs that the implementation is not prepared to accept; but, as the implementation can add some causality, we should also consider the inputs that will become enabled after producing some outputs, i.e. $\exists ?\omega \in \text{poss}(\bot_t \textbf{ after } \omega) : \text{cc}(?\omega, \bot_t \textbf{ after } \omega) \wedge \forall ?\omega' \in \text{poss}(\bot_i \textbf{ after } \omega) : ?\omega \not\sqsubseteq ?\omega'$.

Definition 23 (Blocking because of an input). *Let $i, t \in \mathcal{IOLES}(L)$ and $\omega \in \mathcal{POMSET}(L)$, we have*

$$\textbf{\textit{blocks}}_{\mathcal{I}}(i, t, \omega) \Leftrightarrow \textit{poss}(\bot_t \textbf{ \textit{after}} \ \omega) \not\gg^{+} \textit{poss}(\bot_i \textbf{ \textit{after}} \ \omega)$$

Example 21. Consider implementation i_5 and test case t_5 of Figure 17, the test execution blocks after logging in because $?insurance$ **co** $?train \in \text{poss}(\bot_{t_5}$ **after** $?login)$, but the implementation is not able to accept it (nor any of its refinements), i.e. $\forall ?\omega \in \text{poss}(\bot_{i_5}$ **after** $?login)$ $:?insurance$ **co** $?train \not\sqsubseteq ?\omega$, and $\textbf{blocks}_{\mathcal{I}}(i_5, t_5, ?login)$. If we consider i_6 as the implementation, the test execution also blocks because neither the $?insurance$ **co** $?train$ input action nor its refinements are possible in the implementation and $\textbf{blocks}_{\mathcal{I}}(i_6, t_5, ?login)$.

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

Definition 24 (Failure of a test suite). *Let i be an implementation, and T a test suite, we have:*

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

If the implementation does not fail the test suite, it passes *it, denoted by i **passes** T.*

Example 22. Let $T = \{t_4, t_5\}$ from Figure 17. We have seen in section 4 several situations that lead to the non conformance of an implementation. As seen in Example 20, the execution of the test case t_4 with the (non conforming) implementations i_2, i_7, i_8 leads to a blocking, so we have i_2, i_7, i_8 **fails** T. We saw in Example 21 that the test executions between the implementations i_5, i_6 and the test case t_5 block after logging in, therefore we also have i_5, i_6 **fails** T. We can conclude that T is capable of detecting the non conforming implementations presented in the last section. We can easily check that the (correct) implementations i_1, i_3, i_4 pass T.

5.3 Completeness of the Test Suite

We saw in Example 22 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 reject all, and nothing but, non conformant implementations. A test suite which rejects only non conformant implementations is called *sound*, while a test suite that accepts only conformant implementations is called *exhaustive*. A test suite may not be sound if it contains

a test case which is too strict: for instance, a test case containing two weakly concurrent events which would accept only implementations where these events are concurrent, thus rejecting those ordering the events, even though they are correct w.r.t **co-ioco**. In other words, a sound test suite does not produce false negatives. Conversely, a test suite will not be exhaustive if it is too loose and accepts incorrect implementations. A sound and exhaustive test suite is called *complete*.

Definition 25 (Properties of test suites). *Let s be a specification and T a test suite, then*

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

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

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

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

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

Notice that the trace inclusion required in a) ensures that any possible input in the test case is also possible in the specification.

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 24 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:

$$\begin{array}{ll} & \exists \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i, t, \omega) \\ \text{implies} & \{* \text{ Definition 22 } *\} \\ & \exists \omega \in \text{traces}(t) : \text{out}(\bot_i \textbf{ after } \omega) \not\gg^{-} \text{out}(\bot_t \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 19 } *\} \\ & \exists \omega \in \text{traces}(t) : \exists x \in \text{out}(\bot_i \textbf{ after } \omega) : \forall x' \in \text{out}(\bot_t \textbf{ after } \omega) : \ x' \not\sqsubseteq x \\ \text{implies} & \{* \text{ Assumptions a) and b) } *\} \\ & \exists \omega \in \text{traces}(s) : \exists x \in \text{out}(\bot_i \textbf{ after } \omega) : \forall x' \in \text{out}(\bot_s \textbf{ after } \omega) : \ x' \not\sqsubseteq x \\ \text{implies} & \{* \text{ Definition 19 } *\} \\ & \exists \omega \in \text{traces}(s) : \text{out}(\bot_i \textbf{ after } \omega) \not\gg^{-} \text{out}(\bot_s \textbf{ after } \omega) \\ \text{implies} & \{* \text{ Definition 20 } *\} \\ & \neg(i \textbf{ co-ioco } s) \end{array}$$

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

$$
\begin{array}{ll}
 & \exists \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{I}}(i, t, \omega) \\
\text{implies} & \{* \text{ Definition 23 } *\} \\
 & \exists \omega \in \text{traces}(t) : \text{poss}(\bot_t \textbf{ after } \omega) \not\gg^{+} \text{poss}(\bot_i \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 18 } *\} \\
 & \exists \omega \in \text{traces}(t) : \exists ?\omega \in \text{poss}(\bot_t \textbf{ after } \omega) : \\
 & \text{cc}(?\omega, \bot_t \textbf{ after } \omega) \wedge \forall ?\omega' \in \text{poss}(\bot_i \textbf{ after } \omega) : !\omega \not\sqsubseteq !\omega' \\
\text{implies} & \{* \text{ Assumptions a) and c) } *\} \\
 & \exists \omega \in \text{traces}(s) : \exists ?\omega \in \text{poss}(\bot_s \textbf{ after } \omega) : \\
 & \text{cc}(?\omega, \bot_s \textbf{ after } \omega) \wedge \forall ?\omega' \in \text{poss}(\bot_i \textbf{ after } \omega) : !\omega \not\sqsubseteq !\omega' \\
\text{implies} & \{* \text{ Definition 18 } *\} \\
 & \exists \omega \in \text{traces}(s) : \text{poss}(\bot_s \textbf{ after } \omega) \not\gg^{+} \text{poss}(\bot_i \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 20 } *\} \\
 & \neg(i \textbf{ co-ioco } s)
\end{array}
$$

We can easily see that the test suite $\{t_4, t_5\}$ proposed in Figure 17 satisfies the three properties of Theorem 1 and thus is sound w.r.t the specification s.

The following theorem gives sufficient conditions for the test suite to be exhaustive.

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

a) $\forall \omega \in \textit{traces}(s) : \exists t \in T : \omega \in \textit{traces}(t)$
b) $\forall t \in T, \omega \in \textit{traces}(t) : \textit{out}(\bot_t \textbf{ after } \omega) \subseteq \textit{out}(\bot_s \textbf{ after } \omega)$
c) $\forall t \in T, \omega \in \textit{traces}(t) : \textit{cc}(?\omega, \bot_s \textbf{ after } \omega) \Rightarrow \textit{cc}(?\omega, \bot_t \textbf{ after } \omega)$

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 two cases holds:

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

$$
\begin{array}{ll}
 & \exists \omega \in \text{traces}(s) : \text{out}(\bot_i \textbf{ after } \omega) \not\gg^{-} \text{out}(\bot_s \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 19 } *\} \\
 & \exists \omega \in \text{traces}(s) : \exists x \in \text{out}(\bot_i \textbf{ after } \omega) : \forall x' \in \text{out}(\bot_s \textbf{ after } \omega) : \ x' \not\sqsubseteq \ x \\
\text{implies} & \{* \text{ Assumptions a) and b) } *\} \\
 & \exists t \in T, \omega \in \text{traces}(t) : \exists x \in \text{out}(\bot_i \textbf{ after } \omega) : \forall x' \in \text{out}(\bot_t \textbf{ after } \omega) : \ x' \not\sqsubseteq \ x \\
\text{implies} & \{* \text{ Definition 19 } *\} \\
 & \exists t \in T, \omega \in \text{traces}(t) : \text{out}(\bot_i \textbf{ after } \omega) \not\gg^{-} \text{out}(\bot_t \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 22 } *\} \\
 & \exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{O}}(i, t, \omega) \\
\text{implies} & \{* \text{ Definition 24 } *\} \\
 & i \textbf{ fails } T
\end{array}
$$

2. The implementation does not conform to the specification because an input from the specification is not possible in the implementation (neither its refinements):

$$
\begin{array}{ll}
 & \exists \omega \in \text{traces}(s) : \text{poss}(\bot_s \textbf{ after } \omega) \not\gg^{+} \text{poss}(\bot_i \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 18 } *\} \\
 & \exists \omega \in \text{traces}(s) : \exists ?\omega \in \text{poss}(\bot_s \textbf{ after } \omega) : \\
 & \text{cc}(?\omega, \bot_s \textbf{ after } \omega) \wedge \forall ?\omega' \in \text{poss}(\bot_i \textbf{ after } \omega) : !\omega \not\sqsubseteq !\omega' \\
\text{implies} & \{* \text{ by Assumption c) we can find } t \text{ such that cc}(?\omega, \bot_t \textbf{ after } \omega) \text{ and by} \\
 & \text{Assumption a) } ?\omega \in \text{poss}(\bot_t \textbf{ after } \omega) \ *\} \\
 & \exists t \in T, \omega \in \text{traces}(t) : \exists ?\omega \in \text{poss}(\bot_t \textbf{ after } \omega) : \\
 & \text{cc}(?\omega, \bot_t \textbf{ after } \omega) \wedge \forall ?\omega' \in \text{poss}(\bot_i \textbf{ after } \omega) : !\omega \not\sqsubseteq !\omega' \\
\text{implies} & \{* \text{ Definition 18 } *\} \\
 & \exists t \in T, \omega \in \text{traces}(t) : \text{poss}(\bot_t \textbf{ after } \omega) \not\gg^{+} \text{poss}(\bot_i \textbf{ after } \omega) \\
\text{implies} & \{* \text{ Definition 23 } *\} \\
 & \exists t \in T, \omega \in \text{traces}(t) : \textbf{blocks}_{\mathcal{I}}(i, t, \omega) \\
\text{implies} & \{* \text{ Definition 24 } *\} \\
 & i \textbf{ fails } T
\end{array}
$$

We can see that the test suite $\{t_4, t_5\}$ from Figure 17 also satisfies the conditions of Theorem 2, therefore it is exhaustive and thus complete.

While sufficient conditions for soundness and exhaustiveness of test suites have been given, we need more: in practice, only a finite number of test cases can be executed; hence we need a method to select a finite set of relevant test cases covering as many behaviors as possible (thus finding as many anomalies as possible). 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. Any "complex" behavior is built from such basic behaviors. A criterion allowing to cover once each basic behavior described by the specification is presented by the authors [27] using a proper notion of complete prefixes [41].

6 Test Derivation

We have seen sufficient conditions to ensure the completeness of a test suite. In this section we will explain how to construct a test suite that fulfills such conditions.

6.1 An Algorithm to Construct a Complete Test Suite

We now recall the algorithm to build a test suite in the **ioco** setting [40] to explain the main differences with our test derivation algorithm. In addition we prove that the test suite obtained is complete w.r.t **co-ioco** and analyze the complexity of our approach.

Test Derivation for LTS. In the **ioco** theory, the behavior of a test case is described by a (finite) tree with verdicts (**pass**/**fail**) in the leaves, where in each internal node either one specific input action can occur (also any possible output is accepted), or every outputs and the special action θ can occur. The special label $\theta \notin L \cup \{\delta\}$ is used in a test case to detect quiescent states of an implementation, so it can be thought of as the communicating counterpart of a δ-action.

The test cases are denoted using a process-algebraic notation: ";" denotes action prefix and "+" denotes choice. Moreover, for S a set of states, S **after** a denotes the set of states which can be reached from any state in S via action a. Let S be a non-empty set of states, with initially $S = \{s_0\}$. Then a test case t is obtained from S by a finite number of recursive applications of one of the following three nondeterministic choices:

1. (* terminate the test case *)
   $t :=$ **pass**
2. (* give a next input to the implementation *)
   $t := \quad a; t_a$
   $\quad + \{x; \mathbf{fail} \mid x \in L_{\mathcal{O}}, x \notin \mathrm{out}(S)\}$
   $\quad + \{x; t_x \mid x \in L_{\mathcal{O}}, x \in \mathrm{out}(S)\}$
   where $a \in L_{\mathcal{I}}$ such that S **after** $a \neq \emptyset$, t_a is obtained recursively by applying the algorithm for S **after** a, and for each $x \in \mathrm{out}(S), t_x$ is obtained by recursively applying the algorithm for the set of states S **after** x.
3. (* check the next output of the implementation *)
   $t := \quad \{x; \mathbf{fail} \mid x \in L_{\mathcal{O}}, x \notin \mathrm{out}(S)\}$
   $\quad + \{\theta; \mathbf{fail} \mid \delta \notin \mathrm{out}(S)\}$
   $\quad + \{x; t_x \mid x \in L_{\mathcal{O}}, x \in \mathrm{out}(S)\}$
   $\quad + \{\theta; t_\theta \mid \delta \in \mathrm{out}(S)\}$
   where t_x and t_θ are obtained by recursively applying the algorithm for S **after** x and S **after** δ respectively.

Test Derivation for IOLES. The basic idea of our algorithm is to divide the specification into behaviors triggered by incompatible inputs, that are prefixes of the specification (with a particular property that we call input choice free, to be defined below), and then to build test cases from finite prefixes of these event structures.

We have explained in the previous section the reason to avoid immediate conflict between input events in a test case. Hence we start by dividing the specification in prefixes in a way that any choice between inputs is represented by one of those prefixes. We call such prefixes *input choice free.*

Definition 26 (Input choice free IOLES). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$, *we have*

$$\mathcal{E} \text{ is input choice free} \Leftrightarrow (E^{\mathcal{I}} \times E^{\mathcal{I}}) \cap \overline{\#} = \emptyset$$

Algorithm 1 builds an input choice free IOLES by removing silent actions and 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 4) and conserving concurrency. At the end of the algorithm, all input conflicts have been resolved in one way, following one fixed strategy of resolution of immediate input conflicts. Such a strategy can be represented as a linearization of the causality relation that specifies in which order the events are selected by the algorithm. In order to cover the other branches, the algorithm must be run several times with *different* conflict resolution schemes, i.e. different linearizations, to obtain a test suite that represents every possible event in at least one test case. However, as it can be seen in Section 6.3, the number of linearizations needed is bounded by the number of direct conflicts between inputs.

Definition 27 (Linearization of a partial order). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L)$. A total order $\mathcal{R}$ over E is a linearization of $\leq$ if for all $e, e' \in E$ we have that $e \leq e'$ implies $e\mathcal{R}e'$.*

Analogous to the non deterministic choice of the next input in the algorithm for LTSs (point 2.), assume a linearization is selected non deterministically. The algorithm for LTSs builds the test case that accepts any output the implementation may produce (points 2. and 3.) returning a **pass** verdict if the output was specified and a **fail** one if not. Contrary to this, we build a test case that only allow to accept those outputs that were specified. Finally, the algorithm for LTSs allows to chose for termination (point 1.) while the termination of our algorithm is given by the finiteness of the IOLES that is given as an input to the algorithm (see Theorem 3).

Given a deterministic specification that satisfies Assumption 4 and a linearization of its causality relation, Algorithm 1 constructs an input choice free prefix of the specification as can by seen in Example 23. It is worth noticing that the algorithm does not terminate if the event structure is infinite. However, as it can be seen in Theorem 3, the algorithm is only applied to finite IOLES. A parameter can be added to stop the algorithm at a given depth of the specification, customized by the user [27].

6.2 IICS set

As it is explained above, we need first to be sure that the collection of linearizations that we use considers all resolutions of immediate input conflicts, i.e. is rich enough to provide, for any given immediate input conflict, a pair of linearizations that reverses the order on that pair.

Definition 28. *Fix $\mathcal{E} \in \mathcal{IOLES}(L)$, 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 $e_1 \overline{\#} e_2$, there exist $\mathcal{R}_1, \mathcal{R}_2 \in \mathcal{L}$ such that $e_1 \mathcal{R}_1 e_2$ and $e_2 \mathcal{R}_2 e_1$.*

Proposition 2. *Let $\mathcal{L}$ be an iics set for $\mathcal{E}$ and $e \in E$ with $\lambda(e) \neq \tau$. There exists $\mathcal{R} \in \mathcal{L}$ such that e belongs to the set of events of an IOLES p constructed by Algorithm 1 and $\mathcal{R}$.*

Algorithm 1 Calculate an input choice free prefix of a given event structure

Require: $s = (E, \leq, \#, \lambda) \in \mathcal{IOLES}(L) : \forall e \in E_s^{\mathcal{I}}, e' \in E_s^{\mathcal{O}} : \neg(e \overline{\#}_s e')$, a linearization $\mathcal{R}$ of $\leq$

Ensure: An input choice free prefix of s

$E_p := \emptyset$
$E_{temp} := E$
while $E_{temp} \neq \emptyset$ **do**
  $e_m := \min_{\mathcal{R}}(E_{temp})$ /* the minimum always exists as $\mathcal{R}$ is total and finite */
  $E_{temp} := E_{temp} \setminus \{e_m\}$
  if $(\{e_m\} \times E_p^{\mathcal{I}}) \cap \overline{\#} = \emptyset \wedge (\langle e_m \rangle \backslash E^{\tau}) \subseteq E_p \wedge \lambda(e_m) \neq \tau$ **then**
    /* the current event e_m is not in conflict with any event of the prefix, it is not a silent event and its past (not considering silent events) is already in the prefix */
    $E_p := E_p \cup \{e_m\}$
  end if
end while
$\leq_p := \leq \cap (E_p \times E_p)$
$\#_p := \# \cap (E_p \times E_p)$
$\lambda_p := \lambda_{|E_p}$
return $p = (E_p, \leq_p, \#_p, \lambda_p)$

Proof. Let p be the IOLES constructed by a fix linearization $\mathcal{R}_1$. Suppose e is not in p; then either *(i)* $e \in E^{\mathcal{I}}$ and $\{e\} \times E_p^{\mathcal{I}} \cap \overline{\#} \neq \emptyset$ or *(ii)* $[e \backslash E^{\tau}] \not\subseteq E_p$. In case *(i)*, there exists $e' \in E_p^{\mathcal{I}}$ such that $e \overline{\#} e'$ and $e' \mathcal{R}_1 e$. As $\mathcal{L}$ is an iics set, we know there exist $\mathcal{R}_2 \in \mathcal{L}$ such that $e \mathcal{R}_2 e'$ and then we can use $\mathcal{R}_2$ to construct p' with e belonging to its events. If *(ii)* holds, then there exists $e' \in [e]$ such that $\{e'\} \times E_p^{\mathcal{I}} \cap \neq \emptyset$, and the analysis is analogous to the one in *(i)*.

The following result shows how to construct a test case for the specification from an input choice free prefix of it.

Proposition 3. *Let p be any input choice free prefix of s. If t is a finite prefix of p, then t is a test case.*

Proof. Let t be a finite prefix of p. We need to prove that t is deterministic, that there is no immediate conflict between its input events and that it is finite.

1. As p is input choice free, there is no immediate conflict between its input actions. This is also the case in t as it is its prefix.
2. Its finiteness is immediate from the hypothesis.
3. As the specification is deterministic, so it is p and therefore t.

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

Theorem 3. *From $\mathcal{PREF}(s)$ and a given iics set $\mathcal{L}$ for s, 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) the outputs following a trace of the test are at least those specified; (3) any concurrent complete set of possible input in the test case is concurrent complete in the specification. (1) Trace inclusion is immediate since the algorithm only removes silent actions and resolves conflicts. (2) For a test t and a trace $\omega \in \text{traces}(t)$, if an output in $\text{out}(\bot_s \textbf{ after } \omega)$ is not in $\text{out}(\bot_t \textbf{ after } \omega)$, it means either that it is in conflict with an input in t, which is impossible by Assumption 4, or that its past is not already in t, which is impossible since ω is a trace of t. (3) As inputs are considered as concurrent complete, if the test case has a concurrent complete possible input that is not concurrent complete in the specification, then either a new input event was introduced (which is not possible as the test case is a prefix of the specification) or because some concurrency had been removed; but this is not possible as only conflicting inputs are removed.

Exhaustiveness: By Theorem 2 we need to prove: (1) every trace is represented in at least one test case; (2) the test case does not produce outputs that are not specified; (3) concurrent complete set of inputs of the specification remain as concurrent complete sets in the test case. (1) Clearly, for all $\omega \in \text{traces}(s)$ there exists at least one prefix $c \in \mathcal{PREF}(s)$ such that $\omega \in \text{traces}(c)$. By Proposition 2 we can find $\mathcal{R} \in \mathcal{L}$ such that this trace remains in the test case obtained by the algorithm. (2-3) The inclusion of outputs and preservation of concurrent complete sets is immediate since the algorithm does not add events.

Example 23. Let $\mathcal{R}_1 = ?login \rightarrow !us_data \rightarrow ?insurance \rightarrow !ins_price \rightarrow !ins_data \rightarrow ?plane \rightarrow !p_price \rightarrow ?train \rightarrow \tau \rightarrow !t_price2 \rightarrow !t_price1$ and $\mathcal{R}_2 = ?login \rightarrow !us_data \rightarrow ?insurance \rightarrow !ins_price \rightarrow !ins_data \rightarrow ?train \rightarrow \tau \rightarrow !t_price2 \rightarrow !t_price1 \rightarrow ?plane \rightarrow !p_price$ two total orders of the events of s. Both $\mathcal{R}1$ and $\mathcal{R}_2$ are linearizations of $\leq_s$ and form an iics set of s. The input choice free prefix t_4 can be obtained by the Algorithm 1 using $\mathcal{R}1$ while t_5 is obtained with $\mathcal{R}2$. As s is finite, by Theorem 3 we have that $\{t_4, t_5\}$ is a complete test suite.

6.3 Upper Bound for the Complexity of the Method

The complexity of constructing a complete test suite depends on the size of the iics set $\mathcal{L}$ used: for a finite prefix of the system, we need one test case for each linearization in $\mathcal{L}$. We present an upper bound for the size of $\mathcal{L}$ and discuss informally how to improve on it.

Consider linearizations $\mathcal{R}_1$ and $\mathcal{R}_3 = ?login \rightarrow ?plane \rightarrow !p_price \rightarrow ?train \rightarrow \tau \rightarrow !t_price2 \rightarrow !t_price1 \rightarrow ?insurance \rightarrow !ins_price \rightarrow !ins_data \rightarrow !us_data$. We can easily see that some events of them commute, however, Algorithm 1 constructs t_4 whichever of them we use. We have seen that some commutations of events in the linearization produce different test cases (as it is the case of $\mathcal{R}_2$ and t_5). The concept of partial commutation was introduced by Mazurkiewicz [42] where he defines a *trace* as a congruence of a word (or sequence) modulo identities of the form $ab = ba$ for some pairs of letters.

Let Σ be a finite alphabet (its elements are called letters) and $I \subseteq \Sigma \times \Sigma$ a symmetric and irreflexive relation called *independence* or *commutation*. The complement of I is called the *dependence* relation D. The relation I induces an equivalence relation $\equiv_I$ over Σ^*. Two words x and y are equivalent ($x \equiv_I y$) if there exists a sequence $z_1, \ldots, z_k$ of words such that $x = z_1, y = z_k$ and for all $1 \leq i \leq k$ there exists words z_i', z_i'' and letters a_i, b_i satisfying

$$z_i = z_i' a_i b_i z_i'', \qquad z_{i+1} = z_i' b_i a_i z_i'', \text{ and } (a_i, b_i) \in I$$

Thus, two words are equivalent by $\equiv_I$ if one can be obtained from the other by successive commutation of neighboring independent letters.

For a word $x \in \Sigma^*$ the equivalence class of x under $\equiv_I$ is defined as $[x]_I \triangleq \{y \in \Sigma^* \mid x \equiv_I y\}$.

Example 24. Consider $\Sigma = \{a, b, c, d\}$ and $I = \{(a, d)(d, a)(b, c)(c, d)\}$, we have:

$$[baadcb]_I = \{baadcb, baadbc, badacb, badabc, bdaacb, bdaabc\}$$

As explained above, several linearizations of the causality relation build the same test case, therefore they can be seen as equivalent under some relation and we only need one representative for each class. It is shown by Rozenberg and Salomaa [43] that every (Mazurkiewicz's) trace has a unique normal form (every trace in the equivalence class has the same one) and an algorithm is given to construct it.

We have seen that the order between concurrent events or output events in immediate conflict do not change the test cases constructed by Algorithm 1, but immediate conflict between inputs and causality does. We propose the following independence relation:

$$I_s \triangleq (E \times E) \backslash (\leq \cup (\overline{\#} \cap E^{\mathcal{I}} \times E^{\mathcal{I}}))$$

For constructing a test case, we can consider only the normal form of all the possible linearizations (one representative per equivalence class) and therefore the cardinality of the test suite is bounded by the number of equivalence classes under $\equiv_{I_s}$.

Lemma 1. *Let $\mathcal{K} = |\overline{\#} \cap (E^{\mathcal{I}} \times E^{\mathcal{I}})|$, then Algorithm 1 needs to be run only $2^{\mathcal{K}}$ times to obtain a complete test suite.*

Example 25. Consider system s_5 from Figure 18 and $e_2, e_3, e_4 \in E^{\mathcal{I}}$. The dependence relation contains all the pairs of events that are related either by causality or input immediate conflict. Now consider $\mathcal{R} = \perp e_1 e_2 e_8 e_3 e_6 e_4 e_5 e_7$ and $\mathcal{R}' = \perp e_1 e_8 e_2 e_5 e_3 e_6 e_4 e_7$ two linearizations of $\leq_s$. The Foata normal form of both $\mathcal{R}$ and $\mathcal{R}'$ is $(\perp)(e_1)(e_2)(e_3)(e_4)(e_5 e_6 e_7 e_8)$, meaning that the order of e_5, e_6, e_7, e_8 is not really important for constructing the test case. For any linearization of $\leq_{s_5}$, its normal form is one of the followings:

$$\mathcal{R}_1 = (\perp)(e_1)(e_2)(e_3)(e_4)(e_5 e_6 e_7 e_8) \; \mathcal{R}_2 = (\perp)(e_1)(e_2)(e_4)(e_3)(e_5 e_6 e_7 e_8)$$
$$\mathcal{R}_3 = (\perp)(e_1)(e_3)(e_2)(e_4)(e_5 e_6 e_7 e_8) \; \mathcal{R}_4 = (\perp)(e_1)(e_3)(e_4)(e_2)(e_5 e_6 e_7 e_8)$$
$$\mathcal{R}_5 = (\perp)(e_1)(e_4)(e_2)(e_3)(e_5 e_6 e_7 e_8) \; \mathcal{R}_6 = (\perp)(e_1)(e_4)(e_3)(e_2)(e_5 e_6 e_7 e_8)$$

However, linearizations $\mathcal{R}_1$ and $\mathcal{R}_2$ lead to the same test case (the same happens for $\mathcal{R}_3, \mathcal{R}_4$ and $\mathcal{R}_5, \mathcal{R}_6$). This is due to the fact that once we add an input event to the test case, all the other inputs that are in immediate conflict with it will not be added, and their order is irrelevant. In the example above, linearizations $\mathcal{R}_1, \mathcal{R}_3$ and $\mathcal{R}_5$ construct a complete test suite.

7 Conclusion and Future Work

We have presented a formal framework for conformance testing over *concurrent* systems whose behavior is given in the form of labeled event structures. We propose to distinguish weak and strong concurrency in the specification. Along with the definition of the conformance relation **co-ioco** designed for such specifications, we have defined test cases and test executions, and proposed a test case generation algorithm able to produce a complete test suite. This continues our previous work [26, 27].

Future work includes to handle non-determinism in the specification, and drop Assumption 4 that avoids conflicts between inputs and outputs. One way to avoid making such assumptions would be to assume a fair scheduler. Otherwise, controllability of test cases must be ensured during their construction [36], and the linearizations that are needed to build the test cases should not only reverse the order between conflicting inputs, but also between conflicting inputs and outputs.

All notions of this article are defined in terms of events structures, which is the semantic model for several formalisms. However, real specifications are usually given in one of these *generator* formalisms, such as Petri nets or networks of automata, rather than as event structures. Our approach therefore needs to come on top of an *unfolding* mechanism that generates event structure semantics. An algorithm for building test cases is given by the authors [27] based on an unfolding algorithm. An implementation of this algorithm and the one presented in this article is planned based on the MOLE tool [44], which builds a complete finite prefix of the unfolding of a net.

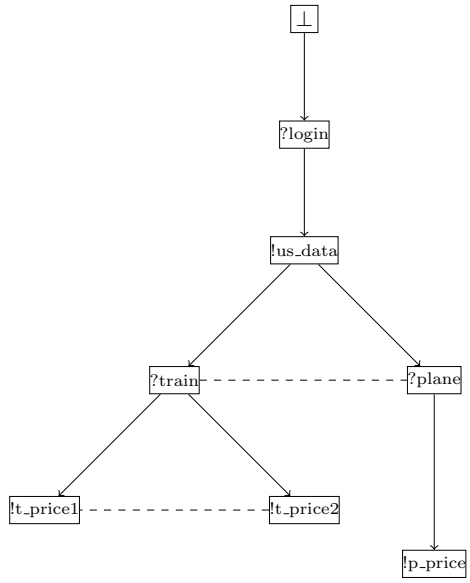
Another important dimension to be explored is *distribution* of observation and of testing. The **dioco** and associated relations studied by Hierons et al. [19, 45] allow to link the conformance of *local* observations to the *global* conformance of the SUT. There, the underlying specification is a multi-port IOTS; by contrast, we shall be studying multi-component, concurrent systems with local observation, and distributed test suites to be developed. In another line of work, Longuet [46] studies different ways of globally and locally testing a distributed system specified with Message Sequence Charts, by defining global and local conformance relations, for which exhaustive test sets are built. Moreover, conditions under which local testing is equivalent to global testing are established under trace semantics. We are currently working on a generalization of those ideas.

References

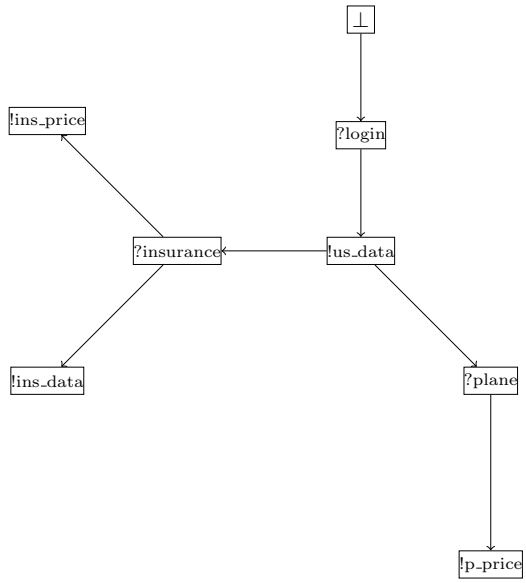
1. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)
2. Hoare, T.: Communicating Sequential Processes. Prentice-Hall (1985)
3. ITU-TS: Recommendation Z.100: Specification and Description Language (2002)
4. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS 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, L., Tretmans, J.: Refusal testing for classes of transition systems with inputs and outputs. In: Formal Techniques for Networked and Distributed Systems. Volume 107 of IFIP Conference Proceedings. (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. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Formal Methods and Testing. Volume 4949 of Lecture Notes in Computer Science., Springer (2008) 77–117
19. 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
20. Hennessy, M.: Algebraic Theory of Processes. MIT Press (1988)
21. Peleska, J., Siegel, M.: From testing theory to test driver implementation. In: Formal Methods Europe. Volume 1051 of Lecture Notes in Computer Science., Springer (1996) 538–556
22. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. 1st edn. John Wiley & Sons, Inc., New York, NY, USA (1999)
23. Ulrich, A., König, H.: Specification-based testing of concurrent systems. In: Formal Techniques for Networked and Distributed Systems. Volume 107 of IFIP Conference Proceedings. (1997) 7–22

24. 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
25. 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
26. Ponce de León, H., Haar, S., Longuet, D.: Conformance relations for labeled event structures. In: Tests and Proofs. Volume 7305 of Lecture Notes in Computer Science., Springer (2012) 83–98
27. Ponce de León, H., Haar, S., Longuet, D.: Unfolding-based test selection for concurrent conformance. In: International Conference on Testing Software and Systems. Lecture Notes in Computer Science, Springer (2013) To appear.
28. Kuske, D., Morin, R.: Pomsets for local trace languages. Journal of Automata, Languages and Combinatorics **7**(2) (2002) 187–224
29. Mokhov, A., Yakovlev, A.: Conditional partial order graphs: Model, synthesis, and application. IEEE Transactions on Computers **59**(11) (2010) 1480–1493
30. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Analysing unlinkability and anonymity using the applied pi calculus. In: Computer Security Foundations, IEEE Computer Society (2010) 107–121
31. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science **13** (1981) 85–108
32. Langerak, R., Brinksma, E.: A complete finite prefix for process algebra. In: Computer Aided Verification. Volume 1633 of Lecture Notes in Computer Science., Springer (1999) 184–195
33. Winskel, G.: Event structures. In: Advances in Petri Nets. Volume 255 of Lecture Notes in Computer Science., Springer (1986) 325–392
34. 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., Springer (1986) 1–20
35. Genc, S., Lafortune, S.: Distributed diagnosis of discrete-event systems using petri nets. In: International Conference on Applications and Theory of Petri Nets. Volume 2679 of Lecture Notes in Computer Science., Springer (2003) 316–336
36. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer **7** (2005) 297–315
37. Xu, Y., Stevens, K.S.: Automatic synthesis of computation interference constraints for relative timing verification. In: International Conference on Computer Design, IEEE (2009) 16–22
38. Heerink, A.W.: Ins and Outs in Refusal Testing. PhD thesis, Universiteit Twente, Enschede (May 1998)
39. Jard, C., Jéron, T., Tanguy, L., Viho, C.: Remote testing can be as powerful as local testing. In: Formal Methods for Protocol Engineering and Distributed Systems. Volume 156 of IFIP Conference Proceedings., Kluwer (1999) 25–40
40. Tretmans, J.: Model based testing with labelled transition systems. In: Formal Methods and Testing. Volume 4949 of Lecture Notes in Computer Science., Springer (2008) 1–38
41. Esparza, J., Römer, S., Vogler, W.: 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., Springer (1996) 87–106
42. Diekert, V., Rozenberg, G., eds.: The Book of Traces. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1995)

43. Rozenberg, G., Salomaa, A., eds.: Handbook of formal languages, vol. 3: beyond words. Springer-Verlag New York, Inc., New York, NY, USA (1997)
44. Schwoon, S.: MOLE. `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`
45. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations and test generation for systems with distributed interfaces. Distributed Computing **25**(1) (2012) 35–62
46. Longuet, D.: Global and local testing from message sequence charts. In: Symposium on Applied Computing, Software Verification and Testing track, ACM (2012) 1332–1338

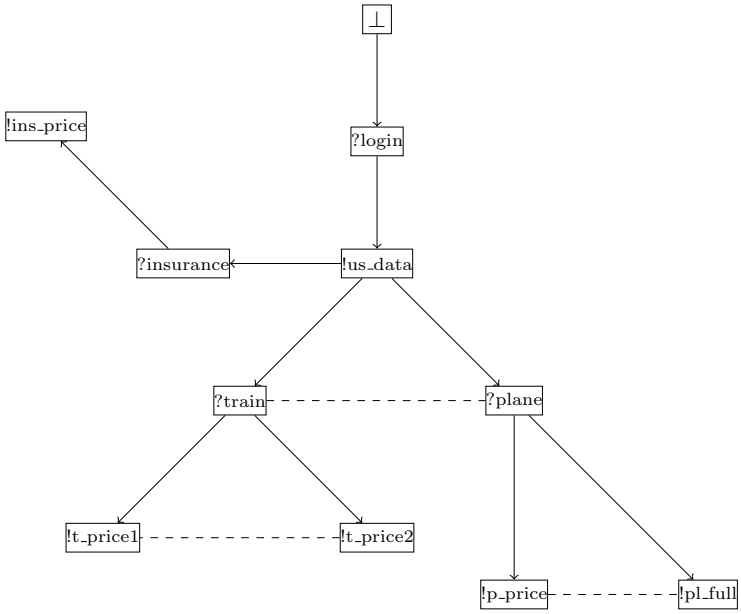


i_5



i_6

Fig. 13. Refused inputs.



i_7

Fig. 14. Extra/Incomplete Outputs.

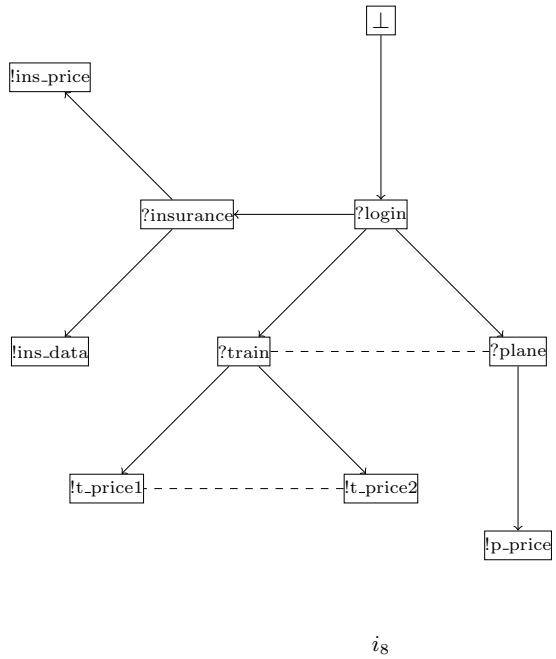


Fig. 15. Extra Quiescence.

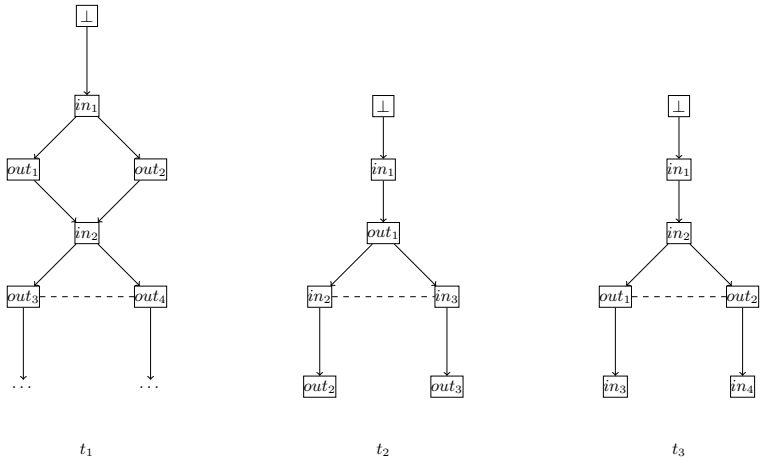
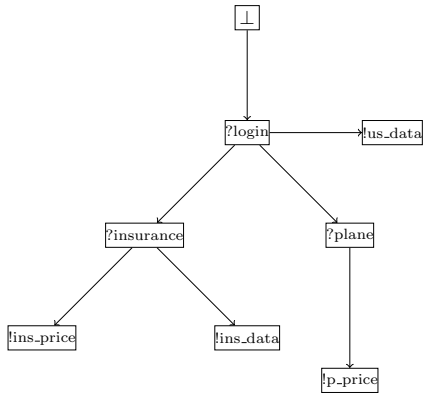
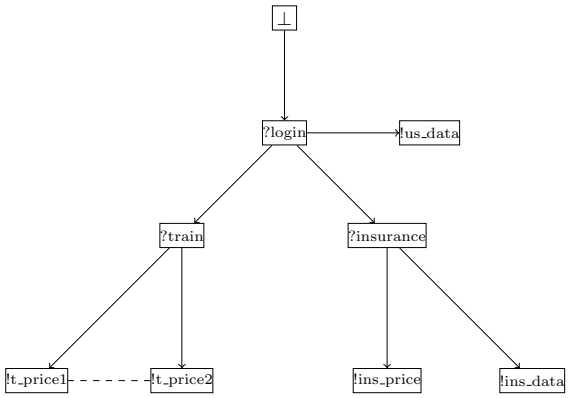


Fig. 16. Left: an infinite event structure, **Center**: immediate conflict between inputs, **Right**: a test case



t_4



t_5

Fig. 17. Two test cases for the travel agency in Figure 4

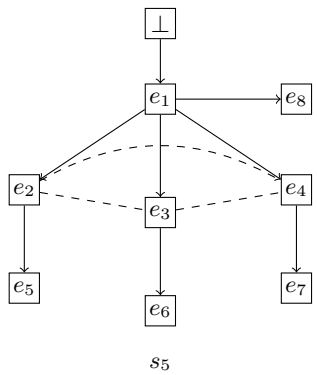


Fig. 18. An IOLES with three inputs in immediate conflict