# Testing concurrent systems through event structures

Hernan Ponce de León

# Testing Concurrent Systems Through Event Structures

Thèse de doctorat

présentée par

Hernán Ponce de León

à

l'École Normale Supérieure de Cachan

en vue de l'obtention du grade de

Docteur en Informatique

et soutenue à Cachan le 7 Novembre 2014 devant le jury composé de :

| Rapporteurs | Examinateurs | Directeurs de thèse |
|---|---|---|
| Rob Hierons | Thierry Jéron | Stefan Haar |
| Alex Yakovlev | Rémi Morin | Delphine Longuet |
| | Pascal Poizat | |



Laboratoire Spécification et Vérification (LSV) ; École Normale Supérieure de Cachan, CNRS & INRIA
61, avenue du Président Wilson ; 94235 CACHAN Cedex, France

# Abstract

Complex systems are everywhere and are part of our daily life. As a consequence, their failures can range from being inconvenient to being life-threatening. Testing is one of the most widely accepted techniques (especially in industry) to detect errors in a system. When the requirements of the system are described by a formal specification, conformance testing is used to guarantee a certain degree of confidence in the correctness of an implementation; in this setting a conformance relation formalizes the notion of correctness. This thesis focuses on conformance testing for concurrent systems.

Conformance testing for concurrent system has mainly focused on models that interpret concurrency by interleavings. This approach does not only suffer from the state space explosion problem, but also lacks the ability to test properties of the specification such as independence between actions. For such reasons, we focus not only on partial order semantics for concurrency, but also propose a new semantics that allows to interleave some actions while forcing others to be implemented as independent.

We propose a generalization of the **ioco** conformance relation, based on Petri nets specifications and their partial order semantics given by their unfoldings, preserving thus independence of actions from the specification. A complete testing framework for this conformance relation is presented. We introduce the notion of global test cases which handle concurrency, reducing not only the size of the test case, but also the number of tests in the test suite. We show how global test cases can be constructed from the unfolding of the specification based on a SAT encoding and we reduce the test selection problem to select a finite prefix of such unfolding: different testing criteria are defined based on the notion of cut-off events.

Finally, we show that assuming each process of a distributed system has a local clock, global conformance can be tested in a distributed testing architecture using only local testers without any communication.

# Résumé

Les systèmes logiciels complexes sont omniprésents dans notre vie quotidienne. De ce fait, un dysfonctionnement peut occasionner aussi bien une simple gêne qu'un danger mettant en péril des vies humaines. Le test est l'une des techniques les plus répandues (en particulier dans l'industrie) pour détecter les erreurs d'un système. Lorsque le cahier des charges d'un système est décrit par une spécification formelle, le test de conformité est utilisé pour garantir un certain niveau de confiance dans la correction d'une implémentation de ce système ; dans ce cadre, la relation de conformité formalise la notion de correction. Cette thèse se focalise sur le test de conformité pour les systèmes concurrents.

Le test de conformité pour les systèmes concurrents utilise principalement des modèles qui interprètent la concurrence par des entrelacements. Néanmoins, cette approche souffre du problème de l'explosion de l'espace d'états et elle n'offre pas la possibilité de tester certaines propriétés de la spécification telle que l'indépendance entre actions. Pour ces raisons, nous utilisons une sémantique d'ordres partiels pour la concurrence. De plus, nous proposons une nouvelle sémantique qui permet à certaines actions concurrentes d'être entrelacées et en force d'autres à être implémentées indépendamment.

Nous proposons une généralisation de la relation de conformité **ioco** où les spécifications sont des réseaux de Petri et leur sémantique d'ordres partiels est donnée par leur dépliage. Cette relation de conformité permet de préserver l'indépendance, dans l'implémentation, des actions spécifiées comme concurrentes. Nous présentons un cadre de test complet pour cette relation. Nous définissons la notion de cas de test globaux gérant la concurrence, réduisant ainsi non seulement la taille des cas de test mais aussi celle de la suite de tests. Nous montrons comment les cas de test globaux peuvent être construits à partir du dépliage de la spécification en s'appuyant sur une traduction SAT, et nous réduisons le problème de la sélection de tests à la sélection d'un préfixe fini de ce dépliage : nous définissons différents critères de sélection à partir de la notion

d'événement limite (cut-off event).

Enfin, en supposant que chaque processus d'un système distribué possède une horloge locale, nous montrons que la conformité globale peut être testée dans une architecture de test distribuée en utilisant seulement des testeurs locaux ne communiquant pas entre eux.

# Acknowledgments

Stefan Haar introduced me to the research world when I first visited LSV for an internship. His dedication, suggestions and guidance converted the lost student that arrived in 2010 into someone able of having his own ideas, express them and convert them into a PhD thesis. During this three years he provided me with the nicest research environment allowing me to participate on conferences, summer schools and pointing me always in the right direction. I extend my gratitude to Delphine Longuet, my co-advisor, for her patience, for teaching me how an article should be writing and presented, for double-checking every proof, for spending hours with a submission, for understanding me. This thesis is the result of all you had taught me. It has not only be an honor to work with you both, it was really a pleasure.

I sincerely thank the reviewers and the jury: Rob Hierons and Alex Yakovlev for accepting reviewing this thesis and all the comments they gave me; Rémi Morin and Pascal Poizat for the time they have kindly employed on me; and Thierry Jéron not only for all his comments and suggestions for this manuscript, but also for his predisposition for all my questions during this three years.

During my PhD studies I had the chance to work with very nice people: Laura Brandán-Briones and Agnes Madalinski introduce me to interesting and new research topics; I had very nice discussions with Pedro D'Argenio; I want to thank Andrey Mokhov for his time to answer all my questions during the last year; I am also thankful to Stefan Scwhoon for accepting supervising an internship with me and for all the nice ideas during those four months; to Konstantinos Athanasiou, thanks for your enthusiasm during the internship, I hope you enjoyed your time at LSV as much as I enjoyed working with you.

I thank DIGITEO for funding my PhD studies through its project TECSTES, and the MEALS project for supporting many research visits to Argentina.

At LSV I met many people: "merci" Thomas Chatain and Loîg Jezequel for every coffee break and for having the greatest patience with my french; Thida

Iem how was always friendly and efficient; Normann Decker my very first friend in Paris; my friends Mahsa Shirmohammadi, Vincent Cheval, Christoph Haase and Simon Theissing, thank you for every meal (double thanks to Mahsa for this) and all the nice moments we had together; to "The" Cesar, because every discussion about unfoldings seems easy compared to the meaning of life.

To my friends of ENS: Martin, Aleksandra and Simon, Nicola and Sandra, Esti, Lara, Matias, Claudia and Miquel, thank you for every lunch, every coffee, every dinner, every poker and every trip.

Para cada argentino que me hizo sentir como en casa: Alito, Flowers, Rusi, Chueka, Ceci, Peti, Cordero (si, vos tambien ya sos argentino), Ana, Anahi, Presi y a todos los que pasaron. Porque ni la lluvia nos pudo quitar nuestros asados. Gracias Timpa y Papelito por los mejores recuerdos sobre la Bestia.

My deepest appreciation to my parents, Ricardo and Corina, for encouraging me to follow my dreams, to my sister, my grandmother, my uncle, aunts and cousins for all their love.

Finally, all my love goes to Aina, because the last two years would have not be the same without you. Te quiero Bonita.

Hernán Ponce de León
Helsinki, Finland
October 2014

# Contents

# 1 Introduction

## 1.1 Motivation

During the last decades, the complexity of software devices has grown very fast. As systems grow in complexity, so does the difficulty to construct them, and consequently, the likelihood of introducing faults in their design or implementation. Failures in safety-critical systems have shown to have dramatic consequences. The following list shows some of the history's worst software bugs [Gar05].

**1962 - Mariner I space probe:** a $80 million rocket launched by NASA, deviated from its intended path and was destroyed by the mission control over the Atlantic Ocean. NASA attributed the error to an improper translation of a formula into computer code.

**1982 - Soviet gas pipeline:** a trans-Siberian gas pipeline exploded due to a bug introduced by operatives working for the CIA into a Canadian computer system purchased to control the pipeline.

**1985/1987 - Therac-25 medical accelerator:** several medical facilities delivered lethal radiation doses, producing the death of at least five patients due to an error generated by a race condition.

**1988 - Buffer overflow in Berkeley Unix finger daemon:** the first internet worm which infected between 2,000 and 6,000 computers in less than a day by taking advantage of a buffer overflow.

**1988/1996 - Kerberos Random Number Generator:** systems that relied on Kerberos protocol for authentication were trivially hacked by the lack of a proper random seed.

**1990 - AT&T Network Outage:** a bug in the software that controlled the long distance switches left around 60 thousands people without long distance service for nine hours.

**1993 - Intel Pentium floating point divide:** a bug in the Pentium chip that generates erroneous float-point divisions cost Intel $475 million.

**1995/1996 - The Ping of Death:** a lack of sanity checks and error handling in the IP fragmentation reassembly code made it possible to crash a wide variety of operating systems by sending malformed "ping" packets.

**1996 - Ariane 5 Flight 501:** 40 seconds after launched, the rocket was destroyed due to an overflow produced by an incorrect conversion from a 64-bit number to a 16-bit number.

**2000 - National Cancer Institute, Panama City:** a miscalculation of the proper dosage of radiation killed at least eight patients.

**2003 - North American Blackout:** caused by a software bug in the alarm system which left operators unaware of the need to re-distribute power after overloaded transmission lines hit unpruned foliage, which triggered a race condition in the control software.

In order to prevent, as much as possible, the faulty behavior of the system and the damage those faults could cause, it is necessary to check that the system behaves as expected; this process is called validation. To validate a system, the intended behavior of the system needs to be described by a specification which explains what the system must do, but not how this is done. An implementation of such a system is a real and executable piece of software.

There are two complementary techniques that can be used to increase the level of confidence in the correct functioning of systems: verification and testing. While verification aims at proving properties about the system by an abstract mathematical model, testing is performed by executing the real implementation. Verification can give certainty about satisfaction of a certain property, but this only applies on the abstract model: "verification is only as good as its underlying model" [Kat99]. On the other hand, testing is applied directly on the implementation, and therefore it is a useful validation technique when a reliable abstract model is not present. However, as Dijkstra already said, since testing is based on observing only a subset of all possible instances of the system behavior, it is usually incomplete: "testing can show the presence of errors, not their absence" [BR70]. This thesis focuses in testing as a validation technique.

The main problem with testing is that it is both expensive and difficult; it is estimated that, on average, it takes 50% of a project budget [Alb76, Mye04]. Manual testing methods lack solid foundations as their strategies are based on heuristics that may not be always successful. Testing is the most applied validation technique in industry and it has been formalized [Tre92, Gau95, CFP96] to bring more confidence in the testing activity.

A system can be tested at different levels of development process and abstraction. One can differentiate between black-box testing and white-box testing based on the degree of visibility assumed in the implementation [Mye04]. Black-box testing assumes only access to the interface of the implementation and not

to its source-code. In contrast to this, white-box tests are derived based on the internal details of the system. Naturally, the degree of visibility between these two can vary, leading to grey-box testing. Other classifications are proposed in the literature [UL07, AO08] based on levels of abstraction: system testing considers the system as a whole; integration testing checks if several components of the system work together properly; component testing aims to check each component individually; unit testing tests the system at its most fine-grained level. Yet another testing distinction lies on the aspects of the system that need to be tested [UL07]: robustness testing explores how the system reacts to underspecified environments or invalid inputs; performance testing focuses on the timing requirements of the system, i.e. how fast the implementation can perform a task; functional testing is used to detect incorrect behaviors of the system with respect to its functional requirements. In this thesis we focus on functional testing at a system level abstraction with a black-box implementation.

## 1.2 Model-based Testing

When we are dealing with black-box testing, we only have the specification of the system from which all the expected behaviors can be derived and that provides the information to build the test cases. When the specification is described by a formal model, we are in the domain of model-based testing. Figure 1.1 shows a typical model-based testing process which comprises four steps:

- Modeling of the expected behavior of the system
- Generation of a set of test cases
- Execution of the test cases on the implementation
- Analysis of the test results to detect differences between the implementation and its expected behavior in order to decide conformance

We consider the implementation under test as black-box system which can only be stimulated via its interface in order to observe how it responses. We differentiate between the inputs or stimulus proposed by the environment and the outputs or reactions produced by the system.

### 1.2.1 The Expected Behavior

The first step in the model-based testing process is the formalization of the expected behavior of the system and the notion of conformance: "what kind of implementations are considered correct?". We assume that there exists a formal specification of the required behavior that can be expressed using a particular specification language, i.e. $\mathcal{S} \in \mathcal{SPECS}$, where $\mathcal{SPECS}$ is the set of all possible specifications. Our aim is to formally reason about the correctness of a concrete implementation $\mathcal{I}$ w.r.t its specification $\mathcal{S}$. Thus, as a common testing hypothesis [Ber91, Tre92], it is assumed that every implementation can

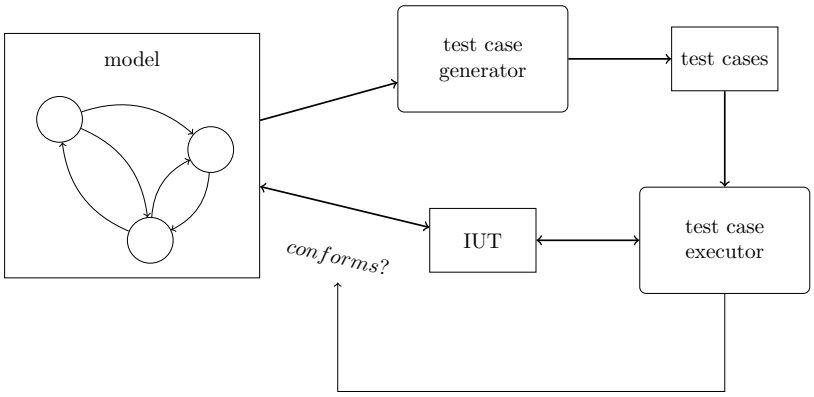



Figure 1.1: Model-based Testing.

be modeled by a formal object $\mathcal{I} \in \mathcal{IMPS}$, where $\mathcal{IMPS}$ denotes the set of all implementation models. Notice that it is not assumed that the model is known, only existence is required. This hypothesis allows us to reason about the (real) implementation as if it was a formal object; conformance can now be expressed as a relation between models and specifications, i.e. $conforms \subseteq \mathcal{IMPS} \times \mathcal{SPECS}$.

There are several models and conformance relations that can be seen as instantiations of $\mathcal{SPECS}$ and $conforms$. [LY96] and [Hie97] use finite state machines as the model of the system and conformance relations relate states of the implementation and specification. When labeled transition systems are used as model of the system, several conformance relation have been proposed, e.g. bisimulation equivalence [Mil90], failure equivalence and preorder [Hoa85], refusals testing [Phi87], observation equivalence [Abr87]. When a distinction is made between input and outputs, the **ioco** conformance relation [Tre92] has become a standard and it is used as a basis in several testing theories for extended state-based models such as restrictive transition systems [HT97, LG05], symbolic transition systems [FGGT08, Jér09], timed transition systems [HLM+08, KT09], multi-port transition systels [HMN08].

### 1.2.2 The Testing Hypotheses

Developing a testing framework without any testing hypothesis is impossible since there always exists an incorrect implementation which is not detected by any test case. As explained in the previous section, every model-based testing framework assumes that the behavior of the implementation can be modeled by some formal object. However there exist several other hypotheses based on the properties one intends to test or the formal object used for modeling the system. When the systems are modeled by finite states machines, [LY96] assumes that the specification does not contain equivalent states, it accepts every input, it is strongly connected and it does not change during the testing procedure; the **ioco**

theory [Tre92] assumes that the system does not contain cycle of unobservable actions, that it is possible to observe when the system blocks (outputs are produced in bounded time) and that the implementation cannot refuse any input proposed by the environment; when the system is distributed [HMN08] assumes that the specification and implementation have the same number of components or interfaces and that not only the implementation, but also the specification is input-enabled.

Some hypothesis like "the implementation can be modeled by some formal object" are necessary: without them conformance is not possible to test; while other hypotheses are practical, but not essential. The latest include the absence of equivalent states or input-enabledness of the implementation.

### 1.2.3 The Experiments

By running experiments where the tester stimulates the system and observes its reactions, one intends to find discrepancies between the implementation and its expected behavior. The behavior of a tester is modeled by a test case $\mathcal{T} \in \mathcal{TESTS}$, where $\mathcal{TESTS}$ denotes the universe of all possible test cases. When the system is modeled by finite state machines [LY96] tests are either sequences of inputs or decision trees (called adaptive sequences) where the input to propose depends on the outputs that the system produced; if systems are modeled by labeled transition systems, test cases are usually represented by labeled transition systems having the form of trees where leaves are special states representing the verdicts of the experiment [Tre08]. The second step of the model-based testing process is the automatic generation of such test cases.

Test cases are executed on the implementation, leading to a verdict that allows to decide non conformance. Formally, a test execution can be modeled by a function $exec : \mathcal{TESTS} \times \mathcal{IMPS} \rightarrow \{\mathbf{pass}, \mathbf{fail}\}$. An implementation $\mathcal{I} \in \mathcal{IMPS}$ passes a test suite $\mathcal{TS} \subseteq \mathcal{TESTS}$ if the test execution of all its test cases leads to a **pass** verdict. If the implementation does not pass the test suite, it fails it. When systems and test cases are modeled by labeled transition systems, test case execution is modeled by synchronous [Tre08] or asynchronous [JJKV98] parallel composition.

## 1.3 Models for Concurrency

This thesis proposes a model-based testing framework for concurrent systems. Various formalisms have been studied for describing the behavior of concurrent and distributed systems and to formally understand their semantics. In addition to traditional models such as languages, automata and labeled transition systems [Plo04], models like Petri nets [Pet62], process algebra [Hoa85, Mil89], Mazurkiewicz traces [Maz88] and event structures [NPW81] have been introduced.

The basic idea behind these formalisms is that they are based on atomic or indivisible units of change (transitions, actions, events or symbols over an

alphabet) which are the basic notions on which computations are built.

One can differentiate between models that allow to explicitly represent the (possibly repeating) states in a system from those which focus instead on the behaviors based on occurrence of actions over time. Examples of the first type are labeled transition systems or Petri nets, while the second type includes reachability trees and event structures. Further distinctions allow to differentiate models that distinguish concurrency seen as independence (for example unfoldings) from nondeterminism and interleavings (for example the reachability graph of a Petri net). Finally, some models represent explicitly all the behaviors of the system while other allow to highlight their branching structure over time. A more detailed classification can be found in [SNW96], where models for concurrency are classified into:

- system or behavior models,
- interleaving or non-interleaving models, and
- linear-time or branching-time models.

Behavior models focus on describing the behavior in terms of the order of atomic actions, abstracting away from states. In contrast, the so-called system models describe explicitly states, which possibly repeat. Interleaving models are those that hide the difference between concurrency between several state machines and nondeterminism inside individual state machines. Non-interleaving models take this difference into account and interpret concurrency as independence. Branching-time models represent the branching structure of the behavior, i.e. the points in which choices are taken, while linear-time models do not.

In this thesis we assume the specification of the system is given by a system model which can be unfolded into a behavior model describing its semantics. We show that independence between actions of the specification cannot be tested when concurrency is interpreted as interleavings and we propose a testing framework based on non-interleaving models.

## 1.4 Model-based Testing of Concurrent Systems

Model-based testing of concurrent systems has been studied for a long time (see for example [Hen88, PS96, Sch99]), however it is most of the time studied in the context of interleaving semantics which is known to suffer from the state space explosion problem. To avoid this problem, Ulrich and König [UK97] 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 from a labeled partial order representing an execution of the behavior machine. Non-interleaving models are also used for generation of test cases in [Hen97, Jar03].

In another direction, Haar et al [HJJ07, vHJJ08] 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 here is an overall complexity reduction, despite the fact that checking partial orders requires in general multiple passes through the same labeled transition to check for presence/absence of specified order relations between input and output events. 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 the system, it is computationally wise to seek alternatives for the direct sequential modeling approach. However, this model still forces to maintain a sequential automaton as the system's skeleton, and to include synchronization constraints (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 approaches presented above avoid the state space problem arising from the implicit representation of all possible interleavings. However, they do not consider that some properties of the specification (for instance independence between actions) may be lost if concurrency is implemented by interleavings.

Example 1.1 (Concurrency as Independence and Interleavings). *Figure 1.2 shows a schematic travel agency that sells services to customers on behalf of different suppliers, some selling tickets (either plane or train ones) and another one selling insurances. In this system, the user can select a ticket and an insurance and the system responses by concurrently producing their corresponding prices. This concurrency (or independence), comes from the fact that each request is served by a different provider.*

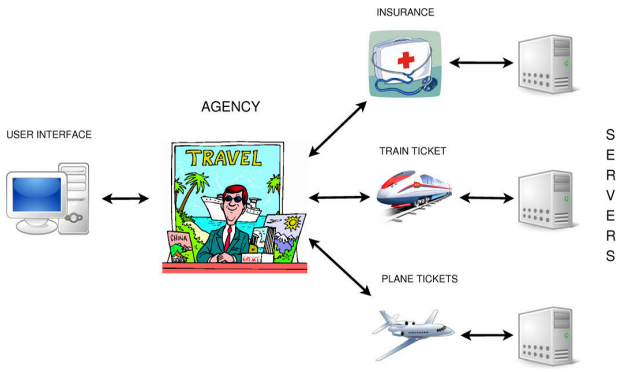



Figure 1.2: A travel agency example.

*Consider the classes (in the object-oriented programming sense) of Figure 1.3 representing the ticket and insurance providers. The interfaces (or specifications) of both providers show a price method that outputs the price of the ticket*

*and insurance based on an initial price (represented by variable* bp*). As it is shown in the bottom left of the figure, the price of the ticket should be independent to the price of the insurance. The implementation of the insurance class is such that every time the user selects an insurance, its price is increased by* 10%*; this is captured by the local variable* f*. The implementation of the ticket provider class has access to the selling policy of the insurance and decides to increase its price based on this information, generating the dependence shown in the bottom right of the figure which was not specified. Besides this extra dependence, if the implementation of the insurance is replaced by another implementation that satisfies the interface, but does not contain variable* f*, the behavior of the system fails as the ticket provider uses this variable, i.e.* Ticket.price: {return bp * Insurance.f}.

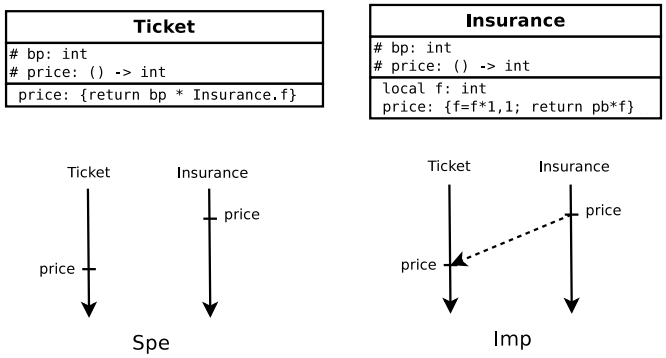



Figure 1.3: Lost of independence in the travel agency.

We want to avoid this kind of extra dependencies. However, if concurrency in the specification is implemented by interleaving, this implementation would be considered correct by most of the conformance relations proposed in the literature and the extra dependence will not be detected. The conformance relation we propose in this thesis allows to distinguish between concurrency seen as independence and interleavings.

Other problems arise when we are dealing with a distributed testing architecture [HU08], i.e. there is one tester placed at each point of control and observation (PCO) and these testers do not communicate. Controllability problems imply that the testers may not know when to apply an input. Figure 1.4 shows a message sequence chart with the interaction of the testers at each PCO with the system under test (SUT). In situation (a) the tester at $PCO_1$ starts with input $?i_1$, observes output $!o_1$ and only after the tester at $PCO_2$ proposes input $?i_2$. Since the tester at $PCO_2$ cannot observe actions at $PCO_1$, it does not know when to propose the input. Another problem that arises in distributed architectures is observability. Consider situation (b) where the tester at $PCO_1$ proposes $?i_1$, observes $!o_1$ before proposing $?i_2$ followed by $!o_2$ and the tester

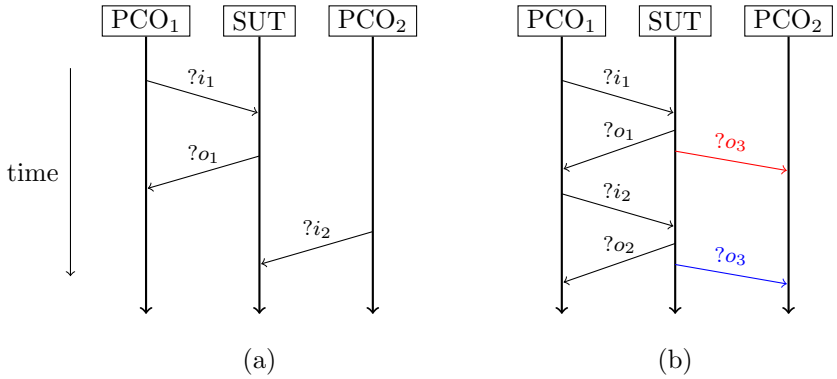



Figure 1.4: Controllability and observability problems in distributed architectures.

at $PCO_2$ observes $!o_3$ (blue arrow) after the four actions in $PCO_1$. From the point of view of the tester at $PCO_2$, only $!o_3$ is observed which is also the case if the system produces $!o_1$ and $!o_3$ (red arrow) at their corresponding PCOs in response to $?i_1$. In this case, two faults have masked one another. Chapter 6 deals with distributed testing architectures and solves controllability and observability problems.

## 1.5 Contributions and Outline

In this thesis, we propose three different formalisms to model the expected behavior of a concurrent system. Labeled transition systems interpret concurrency as interleavings, Petri net as true independence between actions and conditional partial order graphs allow both interpretations based on extra information of the specification. Based on the different semantics, we propose a complete testing framework that includes the definition of conformance relations and algorithms for the generation of test cases. Finally, when the testing architecture is distributed, we show how global conformance can be achieved by local testers.

More specifically, the outline and contributions of the manuscript are:

- Chapter 2. Labeled transition systems and reachability trees are used to model the specification of the system interpreting concurrency as interleaving. Different notions of observations such as traces, outputs, quiescence and refusals are introduced and the **ioco** theory is presented. This is usually done directly on labeled transition systems while we do it on their reachability trees in order to compare **ioco** with the other conformance relations of this thesis. In addition we show how to relax the input-enabledness assumption on the implementation.

- Chapter 3. Petri nets and event structures are used as the specification model which allow to model concurrency explicitly. We extend the stan-

dard notions of traces, produced outputs, quiescence and refusals for partial order semantics and propose the **co-ioco** conformance relation that allows to distinguish concurrency as independence between actions from interleavings.

- Chapter 4. We introduce a new semantics for concurrent systems where some actions specified as concurrent can be implemented in some order (if any) while others need to preserve their concurrency. This new semantics allows to model possible refinements of the system and its distribution. We present partial order graphs which allow to model both types of concurrency, an algorithm to unfold such graphs into event structures and the **wsc-ioco** conformance relation that is based on this semantics. This chapter is based on [PHL14c] and [PM14].

- Chapter 5. We introduce the notion of global test cases (allowing concurrency) and discuss the test execution in the presence of concurrency. We propose an algorithm to derive global test cases from a given specification using a SAT encoding. Finally, we discuss how to select a finite number of test cases, based on finite prefixes of the unfolding, covering as much as possible the behavior of the system. This chapter is based on [PHL13], [PHL14b] and [PHL14c].

- Chapter 6. We describe a distributed testing architecture where the environment cannot globally control and observe the system, but it interacts with it at different points of control and observation. This partial observation of the system usually reduces the testing power, but we show that with extra assumptions on the implementation (using vector clocks), we can reconstruct global traces from partially observed ones allowing to test global conformance locally. This chapter is based on [PHL14a]

- Appendix A. Three other conformance relations are introduced where the interaction between the system and its environment is symmetric (there is no distinction between inputs and outputs). The conformance relations (trace preorder, testing preorder and **conf**) are extensions of the relations with the same names for labeled transition system, but we use event structures as the model of the system with both interleaving and partial order semantics. This chapter is based on [PHL12].

- Appendix B. We present the TOURS prototype that implements the algorithms present in the previous chapters together with a parametric example that is used as a benchmark to our experiments.

## 1.6 Publications

Most of the contributions in this manuscript have already been published in or submitted to international conferences or journals. The following publications partially contain the results of this dissertation:

[**PHL12**] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Conformance relations for labeled event structures. In International Conference on Tests and Proofs, volume 7305 of Lecture Notes in Computer Science, pages 83-98. Springer, 2012.

[**PHL13**] Hernán Ponce de León, Stefan Haar and Delphine Longuet. Unfolding-based test selection for concurrent conformance. In International Conference on Testing Software and Systems, volume 8254 of Lecture Notes in Computer Science, pages 98-113. Springer, 2013.

[**PHL14a**] Hernán Ponce de León, Stefan Haar and Delphine Longuet. Distributed testing of concurrent systems: vector clocks to the rescue. In International Colloquium on Theoretical Aspects of Computing, volume 8687 of Lecture Notes in Computer Science, pages 369-387. Springer, 2014.

[**PHL14b**] Hernán Ponce de León, Stefan Haar and Delphine Longuet. Model-based testing for concurrent systems: Unfolding-based test selection. Software Tools for Technology Transfer, 2014. To appear.

[**PHL14c**] Hernán Ponce de León, Stefan Haar and Delphine Longuet. Model-based testing for concurrent systems with labeled event structures. Software Testing, Verification and Reliability, volume 24(7), pages 558-590, 2014.

During the last three years, the author has also collaborated in the development of two other publications, which have not been integrated into this manuscript to keep the presentation coherent. These are the following:

[**PBB13**] Hernán Ponce de León, Gonzalo Bonigo and Laura Brandán Briones. Distributed Analysis of Diagnosability in Concurrent Systems. In International Workshop on Principles of Diagnosis. 2013.

[**BMP14**] Laura Brandán Briones, Agnes Madalinski and Hernán Ponce de León. Distributed Diagnosability Analysis with Petri Nets. In International Workshop on Principles of Diagnosis. 2014.

# 2

# Conformance Testing for Sequential Models

Languages such as CCS [Mil89], CSP [Hoa85], ACP [BK85] and LOTOS [ISO89], are some of the specification languages that allow to model concurrency between actions. These languages interpret concurrency as a non deterministic choice in the order that actions can occur, known as interleaving semantics for concurrency. This chapter presents a process language that allows to model concurrency together with its operational semantics expressed by labeled transition systems and their reachability trees. Since the conformance relations presented in the following two chapters are defined in terms of behavioral models, we modified the usual presentation of the **ioco** theory by unfolding the labeled transition system into its reachability tree and defining the notions of execution, observation and conformance relation on the reachability tree. Finally we discuss how the usual input-enabledness assumption of the implementation can be dropped.

## 2.1 Labeled Transition Systems

A Labeled Transition System (LTS) is a structure that consists of states and transitions between them which are labeled by actions from an alphabet $L$. States represent the states of the system and the labeled transitions model the actions that the system can perform in those states. Labels in $L$ represent the observable actions of a system; they model the interaction of the system with its environment. As usual with reactive systems, we differentiate actions proposed by the environment (inputs) from those produced by the system (outputs), thus we split the alphabet as $L = \mathcal{I}n \uplus \mathcal{O}ut$ (as usual, input actions are denoted by ? and output actions by !). Internal actions are denoted by the special label $\tau \notin L$, which is assumed to be unobservable from the point of view of the environment.

**Definition** 2.1 (Labeled Transition System). *An LTS is a tuple* $(Q, L, \Delta, q_0)$ *where*

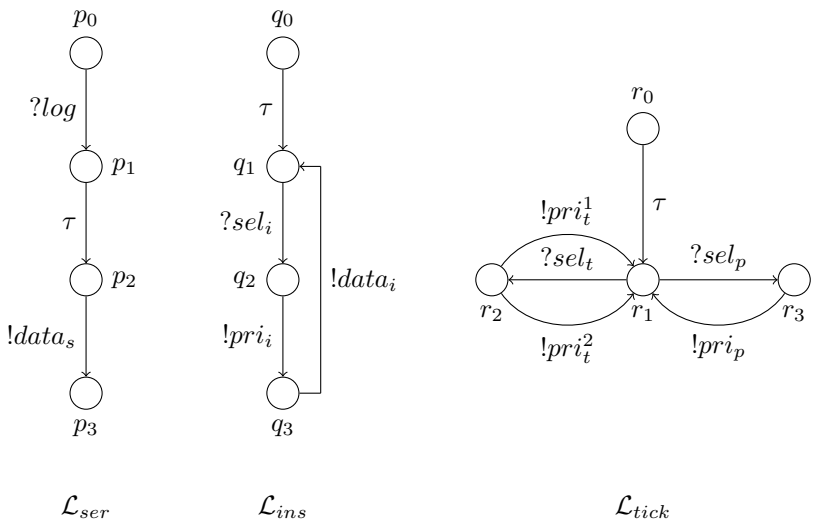



Figure 2.1: Labeled Transition Systems.

- *$Q$ is a countable, non-empty set of states;*
- *$L$ is a countable set of observable actions;*
- *$\Delta \subseteq Q \times (L \cup \{\tau\}) \times Q$, is the transition relation;*
- *$q_0 \in Q$ is the initial state.*

The class of all LTSs with alphabet $L$ is denoted by $\mathcal{LTS}(L)$ and, following the classification that was presented in Section 1.3, it is categorized as a system/interleaving/branching-time model.

Example 2.1 (Labeled Transition Systems). *Figure 2.1 shows three LTSs modeling respectively the behavior of a server, an insurance provider and a ticket provider. After the user logs in into the server ($?log$), the server performs some internal action ($\tau$), which is not observable from the environment, before producing some data of the server ($!data_s$). The behavior of the insurance provider is the following: the system initializes ($\tau$) and it enters a loop where the user can select an insurance ($?sel_i$) and the system responses producing a price and some data of the insurance ($!pri_i$ and $!data_i$). The provider of tickets allows some choices after initializing ($\tau$): the user can select either a plane or a train ticket ($?sel_p$ or $?sel_t$). If a plain ticket is selected, the system produces its price ($!pri_p$), but one out of two possible prices ($!pri_t^1$ or $!pri_t^2$) is produced if a train ticket is selected. After prices are produced, the user has the choice between tickets again.*

The dynamic behavior of an LTS can be expressed by its reachability tree,

another LTS with an acyclic (and usually infinite) structure that highlights the branching produced by choices.

## 2.2 Branching Semantics for LTSs

The notion of execution can be defined (and it is usually the case) directly over LTSs. However, as we intend to compare interleaving and non-interleaving models and we define the notion of partial order executions in terms of a behavior model, we use reachability trees to capture the branching semantics of a LTS.

### 2.2.1 Reachability Trees

Reachability Trees (RTs) are special LTSs with a tree structure (i.e. no cycles), whose states are called nodes. The initial state of the system is the root of the tree; internal nodes are states of the system reachable from the initial state; and arcs can be seen as occurrences of events (labeled by actions) which connect subsequent states. We denote the class of all RTs with alphabet $L$ by $\mathcal{RT}(L)$.

An RT represents the dynamic behavior of a system by means of the actions it performs and therefore we categorize it as a behavior/interleaving/branching-time models. Even if RTs are a subclass of LTSs, we categorize them as behavior models rather than system model. The reason for this is that a node in an RT contains more information than its corresponding state in the LTS: it indicates not only the current state of the system, but also how that state was reached. This information is given by the unique path between the root and the current node, i.e. by the set of events that occurred.

The RT of an LTS can be obtained by the unfolding algorithm presented below, where the $\varphi$ function relates each node of the tree to its corresponding state in the LTS.

**Algorithm 1** Labeled transition systems unfolding

**Require:** a labeled transition system $\mathcal{L} = (Q, L, \Delta, q_0)$
**Ensure:** the reachability tree of $\mathcal{L}$
1: $N = \{n_0\}$ with $\varphi(n_0) = q_0$
2: $\leq = \emptyset$
3: **while** $\exists n \in N, \exists q, q' \in Q, \exists a \in L : \varphi(n) = q \wedge (q, a, q') \in \Delta$ **do**
4: add a fresh node $n'$ to $N$, add $(n, a, n')$ to $\leq$ and set $\varphi(n') = q'$
5: **return** $(N, L, \leq, n_0)$

Example 2.2 (Reachability Trees). *Figure 2.2 shows the initial part of the RT of the ticket provider obtained by applying Algorithm 1 to $\mathcal{L}_{tick}$ in Figure 2.1. Since $\mathcal{L}_{tick}$ contains cycles, $\mathcal{R}_{tick}$ is infinite. It can be seen in yellow that several nodes represent the same state from the LTS, e.g. $\varphi(n_1) = \varphi(n_1') = \varphi(n_1'') = \varphi(n_1''') = r_1$.*

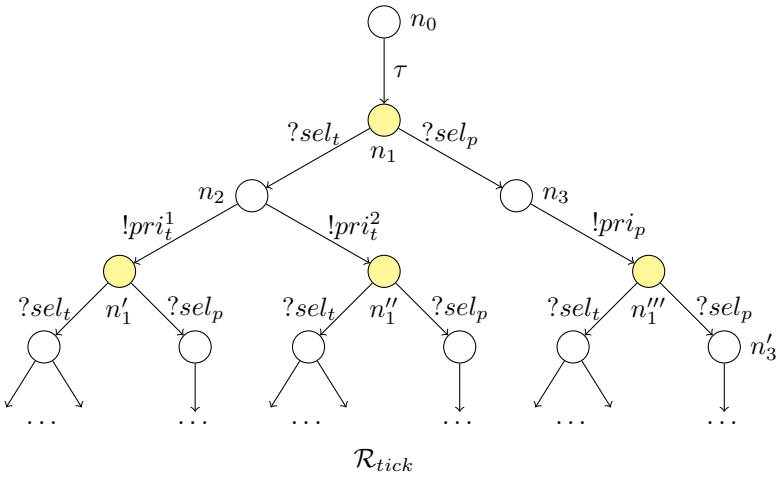



Figure 2.2: Initial part of the reachability tree of the tickets provider.

### 2.2.2 Sequential Executions

The sequential executions of a system are usually described by the sequences of actions it can perform. The following notations are usually defined directly over LTSs, however, we choose to define them over RTs to compare sequential executions and partial order executions defined in the next chapter.

**Definition** 2.2 (Sequential Executions). *Let $(N, L, \leq, n_0) \in \mathcal{RT}(L)$ with $n, n' \in N$ and $\mu, \mu_i \in (L \cup \{\tau\})$ for $i \in [1, k]$, we define*

$$\begin{array}{lcl} n \xrightarrow{\mu} n' & \triangleq & (n, \mu, n') \in \leq \\ n \xrightarrow{\mu_1 \dots \mu_k} n' & \triangleq & \exists n_0, \dots, n_k : n = n_0 \xrightarrow{\mu_1} n_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_k} n_k = n' \\ n \xrightarrow{\mu_1 \dots \mu_k} & \triangleq & \exists n' : n \xrightarrow{\mu_1 \dots \mu_k} n' \end{array}$$

*We say that $\mu_1 \dots \mu_k$ is a sequential execution of $n$ if $n \xrightarrow{\mu_1 \dots \mu_k}$.*

If $\mathcal{R} = (N, L, \leq, n_0) \in \mathcal{RT}(L)$, we will identify the process $\mathcal{R}$ with its initial state $n_0$, i.e. we equally use $\mathcal{R} \xrightarrow{\mu_1 \dots \mu_k}$ and $n_0 \xrightarrow{\mu_1 \dots \mu_k}$.

Example 2.3 (Sequential Executions). *Consider Figure 2.2 and the sequence of actions $\mu = \tau \cdot ?sel_p \cdot !pri_p \cdot ?sel_p$ where $\cdot$ denotes concatenation of sequences; we have $n_0 \xrightarrow{\mu} n_3'$. The same sequence of actions can be followed from the initial state $r_0$ to state $r_3$ in the $\mathcal{L}_{tick}$. This shows the relation between an LTS and its RTs.*

### 2.2.3 Modeling Concurrency with LTSs

LTSs are a powerful semantic model to reason about processes and their interactions. However, for concurrent systems, we need to explicitly represent all the

possible interleavings between concurrent actions and explicit representation by means of a graph or a tree becomes impractical if not infeasible. To overcome this problem, we define a language with LTSs as its operational semantics. Each expression in such a language defines (through its semantics) a LTS. Expressions can be combined by operators in a way that complex behaviors can be composed from simpler ones. We call such a language process language.

The language we use is similar to CCS or CSP. Expressions defining a LTS are called behavioral expressions. We have the following syntax for behavioral expressions, where $a \in L$, $\tau \notin L$ are labels, $B$ is a behavioral expression, $\mathcal{B}$ is a countable set of behavioral expressions, $G \subseteq L$ is a set of labels, and $P$ is a process name.

$$B := \quad a;B \quad | \quad \tau;B \quad | \quad \Sigma\mathcal{B} \quad | \quad B \parallel_G B \quad | \quad \textbf{hide } G \textbf{ in } B \quad | \quad P$$

The sequential expression $a;B$ defines the behavior which can perform $a$ and then behaves as $B$, i.e. the transition system which makes a transition labeled by $a$ to the transition system representing $B$. Expression $\tau;B$ is similar to $a;B$, the difference being that $\tau$ represents an internal action.

The choice expression $\Sigma\mathcal{B}$ behaves as exactly one of the processes in the set $\mathcal{B}$. The choice is determined by the first transition which is made. Whenever the set $\mathcal{B}$ is only composed of two processes $B_1$ and $B_2$, we use the standard notation $B_1 + B_2$.

The parallel expression $B_1 \parallel_G B_2$ represents the concurrent execution of processes $B_1$ and $B_2$. In this concurrent execution, the processes synchronize over all actions in $G$, whereas all actions not in $G$ (including $\tau$) can occur independently in both processes, i.e. they can interleave.

The hiding expression **hide** $G$ **in** $B$ denotes the transition system $B$ where actions $G$ have been hidden, i.e. replaced by the internal action $\tau$.

The last language constructor is process definition, it links a process name to a behavioral expression: $P := B$. The name $P$ can then be used as a process instantiation in behavioral expressions to stand for the behavior contained in its corresponding process definition.

As usual, parentheses are used to disambiguate expressions. If no parentheses are used ";" binds stronger than "$\Sigma$", which binds stronger than " $\parallel_G$ ", which in turn binds stronger than **hide**. The parallel operators are read from left to right, but they are not associative for different synchronization sets.

The semantics for the process language are formally defined in the form of structural operational semantics. Such a semantic definition consists of axioms and inference rules which define, for each behavioral expression, the corresponding labeled transition system (see Table 2.1). Consider as an example the axiom for $a;B$. This axiom is to be read as follows: an expression of the form $a;B$ can always make a transition $\xrightarrow{a}$ to a state from where it behaves as $B$. Consider as another example the inference rule for $\Sigma\mathcal{B}$. Suppose that we can satisfy the premiss, i.e. $B$ can make a transition labeled by $\mu$ to $B'$, $B \in \mathcal{B}$, and $\mu$ is an observable or internal action, then we can conclude that $\Sigma\mathcal{B}$ can make a transition labeled by $\mu$ to $B'$. We give the remaining axioms and rules for our language in Table 2.1.

$$\frac{}{a; B \xrightarrow{a} B} \qquad \frac{}{\tau; B \xrightarrow{\tau} B} \qquad \frac{B \in \mathcal{B} \quad \mu \in L \cup \{\tau\} \quad B \xrightarrow{\mu} B'}{\Sigma\mathcal{B} \xrightarrow{\mu} B'}$$

$$\frac{\mu \in (L \cup \{\tau\}) \backslash G \quad B_1 \xrightarrow{\mu} B_1'}{B_1 \parallel_G B_2 \xrightarrow{\mu} B_1' \parallel_G B_2} \qquad \frac{\mu \in (L \cup \{\tau\}) \backslash G \quad B_2 \xrightarrow{\mu} B_2'}{B_1 \parallel_G B_2 \xrightarrow{\mu} B_1 \parallel_G B_2'}$$

$$\frac{a \in G \quad B_1 \xrightarrow{a} B_1' \quad B_2 \xrightarrow{a} B_2'}{B_1 \parallel_G B_2 \xrightarrow{a} B_1' \parallel_G B_2'} \qquad \frac{P := B_P \quad \mu \in L \cup \{\tau\} \quad B_P \xrightarrow{\mu} B'}{P \xrightarrow{\mu} B'}$$

$$\frac{a \in G \quad B \xrightarrow{a} B'}{\textbf{hide } G \textbf{ in } B \xrightarrow{\tau} \textbf{hide } G \textbf{ in } B'} \qquad \frac{\mu \notin G \quad B \xrightarrow{\mu} B'}{\textbf{hide } G \textbf{ in } B \xrightarrow{\mu} \textbf{hide } G \textbf{ in } B'}$$

Table 2.1: Operational semantics of the process language.

Example 2.4 (Behavioral Language). *Suppose we modify the LTSs from Figure 2.1 by replacing each $\tau$ transition by a synchronization transition $c$ and use* **stop** *as abbreviation for $\Sigma\emptyset$. Behavioral expressions representing the parallel behavior of the server and insurance processes are:*

$$\begin{array}{lcllcl} P & := & p_0 \parallel_{\{c\}} q_0 & q_0 & := & c; q_1 \\ p_0 & := & ?log; p_1 & q_1 & := & ?sel_i; q_2 \\ p_1 & := & c; p_2 & q_2 & := & !pri_i; q_3 \\ p_2 & := & !data_s; \textbf{stop} & q_3 & := & !data_i; q_1 \end{array}$$

*Consider system $P$ which is composed of two parallel processes $p_0$ and $q_0$ and the inference rules from Table 2.1. Process $q_0$ starts by a synchronization action that $p_0$ is not ready to perform yet, so system $P$ can only perform action ?log and behaves as $p_1 \parallel_{\{c\}} q_0$. From this point, both processes can synchronize and behave as $p_2 \parallel_{\{c\}} q_1$. This process satisfies the premises of two inference rules (rules that specify that each process can perform a local action if this action is not a synchronization) and then we have both*

$$(p_2 \parallel_{\{c\}} q_1) \xrightarrow{!data_s} (\textbf{stop} \parallel_{\{c\}} q_1) \xrightarrow{?sel_i} (\textbf{stop} \parallel_{\{c\}} q_2)$$

$$(p_2 \parallel_{\{c\}} q_1) \xrightarrow{?sel_i} (p_2 \parallel_{\{c\}} q_2) \xrightarrow{!data_s} (\textbf{stop} \parallel_{\{c\}} q_2)$$

*which highlights the fact that LTSs interprets concurrency by means of interleavings.*

## 2.3 Observing LTSs

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). Phillips [Phi87], Heerink and Tretmans [HT97] and Lestiennes and Gaudel [LG05] propose conformance relations that in addition consider 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. The latter situation is usually called quiescence following Segala [Seg97]. In this section, we define the notions of traces, refusals and quiescence in the context of LTS.

### 2.3.1 Traces

The observable behavior of the system is captured by sequences of observable actions. Such sequences can be obtained by abstracting internal actions from the executions of the system. If the system can perform from state $n$ a sequence of actions $a \cdot \tau \cdot b \cdot c \cdot \tau$ with $a, b, c \in L$, we have $n \xrightarrow{a \cdot \tau \cdot b \cdot c \cdot \tau}$ and we write $n \stackrel{a \cdot b \cdot c}{\Longrightarrow}$ for the $\tau$-abstracted sequence of observable actions.

**Definition** 2.3 (Observations). *Let* $(N, L, \leq, n_0) \in \mathcal{RT}(L)$ *with* $n, n' \in N$, $a, a_i \in L$ *for* $i \in [1, k]$ *and* $\sigma \in L^*$, *we define*

$$
\begin{array}{lcl}
n \stackrel{\epsilon}{\Longrightarrow} n' & \triangleq & n = n' \text{ or } n \xrightarrow{\tau \ldots \tau} n' \\
n \stackrel{a}{\Longrightarrow} n' & \triangleq & \exists n_1, n_2 : n \stackrel{\epsilon}{\Longrightarrow} n_1 \xrightarrow{a} n_2 \stackrel{\epsilon}{\Longrightarrow} n' \\
n \stackrel{a_1 \ldots a_k}{\Longrightarrow} n' & \triangleq & \exists n_0, \ldots, n_k : n = n_0 \stackrel{a_1}{\Longrightarrow} n_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_k}{\Longrightarrow} n_k = n' \\
n \stackrel{\sigma}{\Longrightarrow} & \triangleq & \exists n' : n \stackrel{\sigma}{\Longrightarrow} n'
\end{array}
$$

*We say that* $\sigma$ *is an observation of* $n$ *if* $n \stackrel{\sigma}{\Longrightarrow}$.

Example 2.5 (Observations). *It have been shown in Example 2.3 that the sequence* $\mu = \tau \cdot ?sel_p \cdot !pri_p \cdot ?sel_p$ *is a sequential execution from the initial node of* $\mathcal{R}_{tick}$, *i.e.* $n_0 \xrightarrow{\mu} n'_3$. *We can abstract the internal action from this execution leading to the following observation:* $n_0 \stackrel{\sigma}{\Longrightarrow} n'_3$ *with* $\sigma = ?sel_p \cdot !pri_p \cdot ?sel_p$.

The traces of $\mathcal{R} \in \mathcal{RT}(L)$, denoted by traces($\mathcal{R}$), are all the sequences of visible actions that $\mathcal{R}$ can perform from its initial state. Traces and reached states from a given state are captured by the following standard definitions.

**Definition** 2.4 (Traces and Reached States). *Let* $\mathcal{R} = (N, L, \leq, n_0) \in \mathcal{RT}(L)$, $\sigma \in L^*$ *and* $n, n' \in N$, *we define*

$$
\begin{array}{lcl}
traces(\mathcal{R}) & \triangleq & \{\sigma \in L^* \mid n_0 \stackrel{\sigma}{\Longrightarrow}\} \\
n \textbf{ \textit{after} } \sigma & \triangleq & \{n' \mid n \stackrel{\sigma}{\Longrightarrow} n'\}
\end{array}
$$

It is worth to notice that the presence of internal actions implies non-determinism, therefore the reachable state after some trace is not necessarily unique.

Example 2.6 (Traces and Reached States). *Example 2.5 shows that* $n_0 \stackrel{\sigma}{\Longrightarrow} n_3'$ *for* $\sigma = ?sel_p \cdot !pri_p \cdot ?sel_p$*, therefore we can conclude that* $\sigma \in traces(\mathcal{R}_{tick})$ *and* $(n_0 \textbf{ \textit{after}} \ \sigma) = \{n_3'\}$*. However, after the empty observation, the system can be either in* $n_0$ *or* $n_1$*, i.e.* $(n_0 \textbf{ \textit{after}} \ \epsilon) = \{n_0, n_1\}$*.*

### 2.3.2 Quiescence and Produced Outputs

With reactive systems, we need to differentiate states where the system can still produce some outputs, and those where the system blocks. The system can block mainly for one of the following reasons [JJ05]: the system is waiting for an input from the environment (output quiescence); the system cannot evolve (deadlock); the system diverges by an infinite sequence of silent actions (livelock). Blockings of the system are captured by the notion of quiescence [Seg97, STS13].

**Definition** 2.5 (Quiescence). *Let* $(N, L, \leq, n_0) \in \mathcal{RT}(L)$*, a node* $n \in N$ *is quiescent iff* $\forall \mu : n \xrightarrow{\mu}$ *implies* $\mu \notin \mathcal{O}ut$*.*

Example 2.7 (Quiescence). *Consider the ticket provider* $\mathcal{R}_{tick}$ *from Figure 2.2. From node* $n_3$*, it is possible to perform output* $!pri_p$ *and therefore* $n_3$ *is not a quiescent node. Contrary to this, node* $n_1$ *only allows inputs actions* $?sel_t$ *and* $?sel_p$*, thus* $n_1$ *is quiescent.*

In the **ioco** theory, the specification of the system is determinized before constructing the test suite. The notion of quiescence is lost after this determinization, therefore the observation of quiescence is usually made explicit by adding self loops labeled by a $\delta$ action in the LTS for every quiescent state before determinization of the specification. When this LTS is unfolded using Algorithm 1, every quiescent state generates a quiescent node $n$ in the RT such that $n \stackrel{\delta}{\Longrightarrow}$.

If the system is not in a quiescent state, then it can produce some output. The produced outputs of an RT in a given node $n$, are the outputs actions that are enabled in $n$ or $\delta$ if $n$ is quiescent. This notion can be generalized for a set of nodes.

**Definition** 2.6 (Produced Outputs). *Let* $(N, L, \leq, n_0) \in \mathcal{RT}(L), n \in N$ *and* $S \subseteq N$*, we define*

$$\begin{array}{lcl} out(n) & \triangleq & \{!o \in \mathcal{O}ut \mid n \stackrel{!o}{\Longrightarrow}\} \cup \{\delta \mid n \stackrel{\delta}{\Longrightarrow}\} \\ out(S) & \triangleq & \bigcup_{n \in S} out(n) \end{array}$$

Example 2.8 (Produced Outputs). *Consider the ticket provider system* $\mathcal{R}_{tick}$ *in Figure 2.2. Node* $n_0$ *neither is quiescent (a* $\tau$ *action can be performed), nor it*

*allows to produce some outputs, while it is shown in Example 2.7 that* $n_1$ *is a quiescent node and then we have* $n_1 \stackrel{\delta}{\Longrightarrow}$. *After observing the empty trace* $\epsilon$, *the system can be either in node* $n_0$ *or node* $n_1$ *and then* $out(\mathcal{R}_{tick}$ ***after*** $\epsilon) = out(\{n_0, n_1\}) = \{\delta\}$. *After selecting a train ticket, the reached node is unique and the system can produce two outputs, i.e.* $out(\mathcal{R}_{tick}$ ***after*** $?sel_t) = out(n_2) = \{!pri_t^1, !pri_t^2\}$.

### 2.3.3 Possible Inputs

It is sometimes argued that a reactive system should not be able to block an input proposed by the environment [Tre08]. A system that cannot reject inputs is called input-enabled.

**Definition** 2.7 (Input-enabledness). *Let* $\mathcal{R} = (N, L, \leq, n_0) \in \mathcal{RT}(L)$, *system* $\mathcal{R}$ *is called input-enabled iff* $\forall n \in N, ?i \in \mathcal{I}n : n \stackrel{?i}{\Longrightarrow}$.

The **ioco** theory assumes the input-enabledness of the implementation. This assumption is made to avoid computation interference [XS09] in the parallel composition between the implementation and the test cases. However, even if many realistic systems can be modeled as an input-enabled system, 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. In order to overcome this problem, we also consider the inputs the system refuses to accept.

Refused inputs can be made explicitly observable by a special action as quiescence is observable by a $\delta$ action (see for example [HT97]). Lestiennes and Gaudel [LG05] 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, or equivalently, every input refused by the implementation should be refused by the specification. This implies the assumption 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 environment proposes. As in the case of outputs, possible inputs can be generalized for a set of states.

**Definition** 2.8 (Possible Inputs). *Let* $\mathcal{R} \in \mathcal{RT}(L), n \in N$ *and* $S \subseteq N$, *we define*

$$\begin{array}{lcl} poss(n) & \triangleq & \{?i \in \mathcal{I}n \mid n \stackrel{?i}{\Longrightarrow}\} \\ poss(\emptyset) & \triangleq & \mathcal{I}n \\ poss(S) & \triangleq & \bigcup_{n \in S} poss(n) \end{array}$$

Remark 1. *Since we consider the possible inputs after some trace and the set of reached states after such a trace can be empty, for technical convenience, we define* $poss(\emptyset)$ *as the set of all inputs. For a detailed explanation, see Section 2.4.*

Example 2.9 (Possible Inputs). *In the tickets provider system* $\mathcal{R}_{tick}$ *from Figure 2.2, the system can reach from its initial state (by a* $\tau$ *action) node* $n_1$ *which enables inputs* $?sel_t$ *and* $?sel_p$*. As the system can reach both* $n_0$ *and* $n_1$ *after an empty observation, we have* $poss(\mathcal{R}_{tick}\ \textbf{\textit{after}}\ \epsilon) = poss(\{n_0, n_1\}) = \{?sel_t, ?sel_p\}$*.*

## 2.4 The ioco Conformance Relation

This section presents the testing hypotheses and conformance relation for conformance testing of labeled transition systems.

**Testing Hypotheses:** We assume that the specification of the system is given as labeled transition system $(Q, L, \Delta, q_0)$ over alphabet $L = \mathcal{I}n \uplus \mathcal{O}ut$ of input and output labels. To be able to test an implementation against such a specification, we make a set of testing assumptions. First of all, we make the usual testing assumption that the behavior of the SUT itself can be modeled by a labeled transition system over the same alphabet of labels. We also assume as usual that the implementation is input-enabled and that the specification does not contain cycles of silent actions to avoid divergence. The end to this section discusses how the latter can be relaxed.

We present now a conformance relation based on the notions of observation presented in this chapter. Informally, the input-output conformance relation [Tre96a] states that an implementation $\mathcal{I}$ conforms to a specification $\mathcal{S}$ iff after an arbitrary experiment extracted from $\mathcal{S}$, the outputs produced by $\mathcal{I}$ are also produced by $\mathcal{S}$. We have chosen to define this relation over a behavior model (RTs) rather than over LTSs, to unify it with the other conformance relations presented in this thesis, which are defined over behavioral models.

**Definition** 2.9 (**ioco**). *Let* $\mathcal{S}, \mathcal{I} \in \mathcal{RT}(L)$ *be respectively the specification and an input-enabled implementation of the system. The* **ioco** *relation is defined as*

$$\mathcal{I}\ \textbf{ioco}\ \mathcal{S} \Leftrightarrow \forall \sigma \in traces(\mathcal{S}) : out(\mathcal{I}\ \textbf{\textit{after}}\ \sigma) \subseteq out(\mathcal{S}\ \textbf{\textit{after}}\ \sigma)$$

Example 2.10 (Removed Outputs). *Consider Figure 2.3 and the ticket provider specification* $\mathcal{S}_1 = \mathcal{R}_{tick}$ *from Figure 2.2. Implementation* $\mathcal{I}_1$ *removes the possibility to produce a second kind of price after selecting a train ticket, i.e.* $out(\mathcal{I}_1\ \textbf{\textit{after}}\ ?sel_t) = \{!pri_t^1\}$*, but since the produced output is specified, i.e.* $out(\mathcal{S}_1\ \textbf{\textit{after}}\ ?sel_t) = \{!pri_t^1, !pri_t^2\}$*, and the rest of the tree is isomorphic to* $\mathcal{S}_1$*, we can conclude that* $\mathcal{I}_1$ **ioco** $\mathcal{S}_1$*.*

Even if **ioco** allows to remove some outputs the system may produce when the specification allows several ones, at least one of them must be implemented to avoid extra quiescence. Extra quiescence and extra outputs are not allowed.

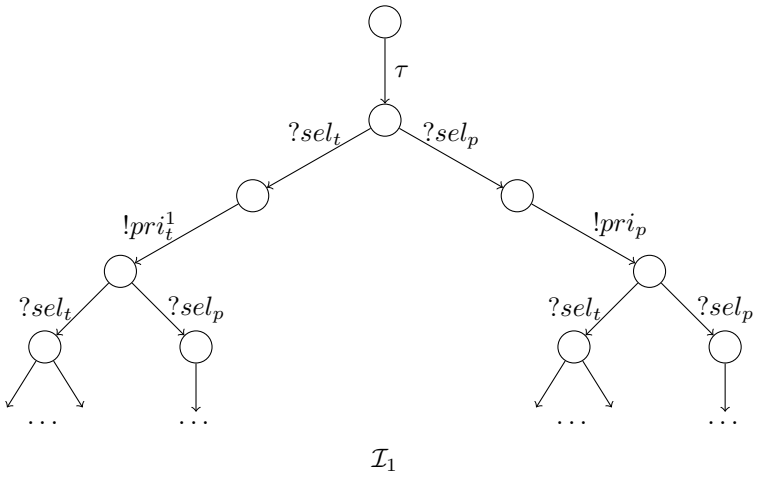



Figure 2.3: A conformant implementation of the ticket provider w.r.t **ioco**.

Example 2.11 (Extra Quiescence and Outputs). *Figure 2.4 shows two implementations with non specified outputs: $\mathcal{I}_2$ allows the user to select a train ticket, but no output is produced. The system reaches a quiescent state after selecting the ticket, i.e. $out(\mathcal{I}_2$ **after** $?sel_t) = \{\delta\}$, which is not a specified output. We have then that $\neg(\mathcal{I}_2$ **ioco** $\mathcal{S}_1)$. Implementation $\mathcal{I}_3$ can produce a second type of price when a plane is selected, $out(\mathcal{I}_3$ **after** $?sel_p) = \{!pri_p, !pri_p^2\}$, but $!pri_p^2 \notin out(\mathcal{S}_1$ **after** $?sel_p)$ so $\neg(\mathcal{I}_3$ **ioco** $\mathcal{S}_1)$.*

The fact that **ioco** only requires inclusion of produced outputs, together with the fact that specifications can be non-input-enabled, makes it possible to have partial specifications. For experiments which are not in traces($\mathcal{S}$), there is no requirement on the implementation, which implies that an implementation is free to implement anything it likes after such a trace.

Example 2.12 (Extra Behaviors). *The implementation presented in Figure 2.5 allows the user to choose a boat ticket rather than a train or plane one. Even if the output $!pri_b$ is not specified, this output is only produced after selecting a boat ticket, and since this input is not specified either, this behavior is never tested. We have $\mathcal{I}_4$* **ioco** *$\mathcal{S}_1$.*

**Avoiding input-enabledness:** An LTS can easily be converted into an input-enabled system by adding loops for any missing input in every state or adding an error state and transitions to this error state for all unexpected inputs. These techniques are known as angelic and diabolic completion [Tre08]. In Figures, we usually avoid drawing unexpected inputs. We propose a conformance relation that drops the input-enabledness assumption of the implementation and test refusals, but which remains conservative w.r.t the standard **ioco**. Consider the

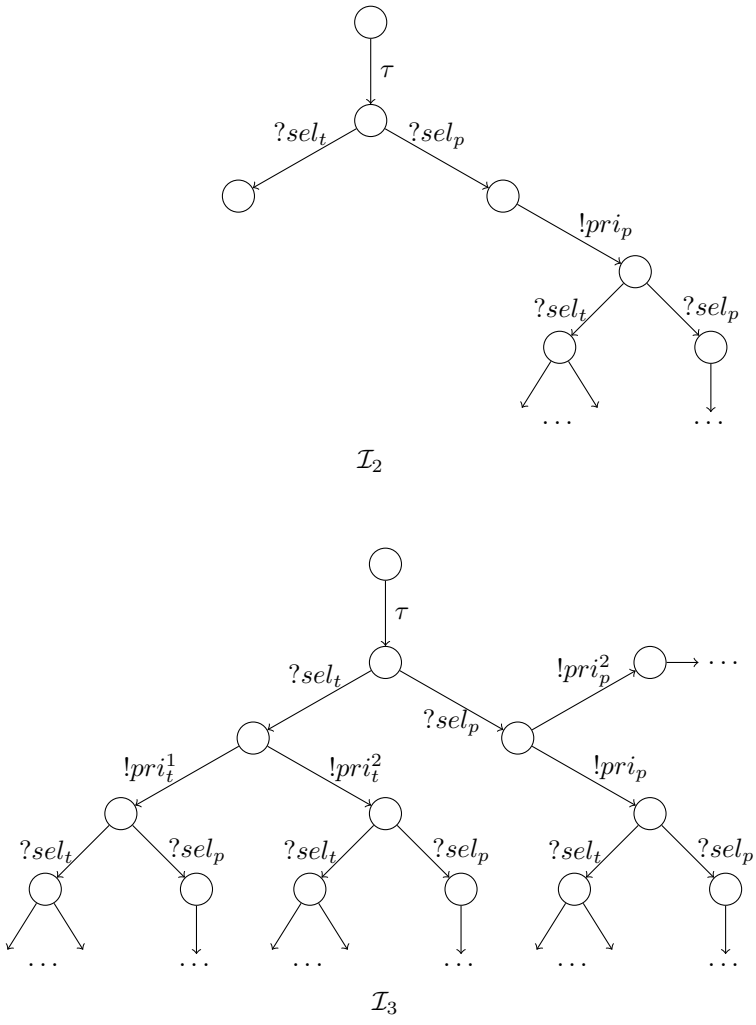



Figure 2.4: Extra quiescence and outputs in the implementation.

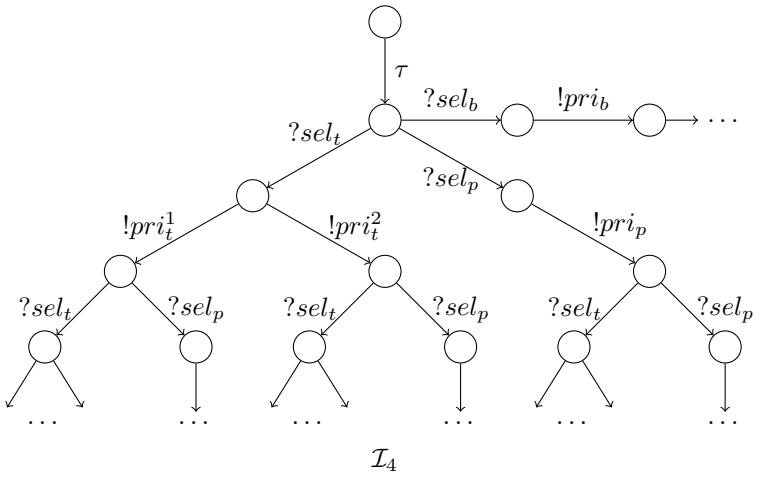



Figure 2.5: Extra behavior in the implementation.

conformance relation below

$$\mathcal{I} \textbf{ ioco}_2\ \mathcal{S} \Leftrightarrow \forall \sigma \in \text{traces}(\mathcal{S}) : \tag{2.1}$$

$$\text{poss}(\mathcal{S} \textbf{ after } \sigma) \subseteq \text{poss}(\mathcal{I} \textbf{ after } \sigma) \tag{2.2}$$

$$\text{out}(\mathcal{I} \textbf{ after } \sigma) \subseteq \text{out}(\mathcal{S} \textbf{ after } \sigma)$$

It is expected that Equation 2.2 reduces to true whenever the implementation is input-enabled. This is not always true if $\text{poss}(\emptyset) \neq \mathcal{I}n$. Suppose we remove $\text{poss}(\emptyset) \triangleq \mathcal{I}n$ from Definition 2.8, and consider $\sigma \in L^*$ which is not a trace of the implementation, then $\text{poss}(\mathcal{I} \textbf{ after } \sigma) = \text{poss}(\emptyset) = \emptyset$. Consider behavioral expressions $\mathcal{S} = ?i_1; (!o_1 + (!o_2; ?i_2))$ and $\mathcal{I} = ?i_1; o_1$. After completing with inputs to make $\mathcal{I}$ input-enabled, we have that $\mathcal{I} \textbf{ ioco } \mathcal{S}$. However, $\neg(\mathcal{I} \textbf{ ioco}_2\ \mathcal{S})$ since $\{?i_2\} = \text{poss}(\mathcal{S} \textbf{ after } ?i_1 \cdot !o_2) \not\subseteq \text{poss}(\mathcal{I} \textbf{ after } ?i_1 \cdot !o_2) = \emptyset$. If $\text{poss}(\emptyset) = \mathcal{I}n$, then Equation 2.2 reduces to true whenever the implementation is input-enabled and **ioco** and $\textbf{ioco}_2$ coincide.

**Theorem** 1. *Let $\mathcal{S}$ and $\mathcal{I}$ be respectively the specification and an input-enabled implementation of a system, then we have*

$$\mathcal{I} \textbf{ ioco } \mathcal{S} \textit{ iff } \mathcal{I} \textbf{ ioco}_2\ \mathcal{S}$$

*Proof.* Suppose $\sigma \in \text{traces}(\mathcal{S})$. If $\sigma \in \text{traces}(\mathcal{I})$, then as the implementation is input-enabled we have $\text{poss}(\mathcal{I} \textbf{ after } \sigma) = \mathcal{I}n$. If $\sigma \notin \text{traces}(\mathcal{I})$, by Definition 2.8 we have $\text{poss}(\mathcal{I} \textbf{ after } \sigma) = \text{poss}(\emptyset) = \mathcal{I}n$. In both cases Equation 2.2 reduces to $\text{poss}(\mathcal{S} \textbf{ after } \sigma) \subseteq \mathcal{I}n$ which is clearly true and both conformance relations coincide. □

To overcome the same problem, [LG05] considers in the conformance relation only traces of the specification that can also be executed in the implementation.

Similar to this, [BJSK12] introduces a refinement relation where inclusion of inputs is required after traces of the implementation and not of the specification. It is possible to avoid defining $\text{poss}(\emptyset) = \mathcal{I}n$ by requiring $\forall \sigma \in \text{traces}(\mathcal{S}) \cap \text{traces}(\mathcal{I})$ instead of (2.1), however we have decided to follow the approach proposed in this thesis to keep our conformance relations as similar as possible to **ioco**.

## 2.5 Conclusion

This chapter introduces the **ioco** theory that allows to test concurrent systems interpreting concurrency by interleavings. The only contribution of this chapter is the definition of the $\mathbf{ioco}_2$ relation that drops the input-enabledness assumption of the implementation and Theorem 1 which shows that under such an assumption **ioco** and $\mathbf{ioco}_2$ coincide.

The presentation we have chosen is different from the traditional one: we show how a labeled transition system can be unfolded into its reachability tree and the notions of executions, observations and conformance relations are defined on the unfolded system. This presentation is the same that we follow in Chapter 3 and Chapter 4 and allows to see the similitudes of the testing framework besides the chosen semantics.

# 3

# Conformance Testing for Non-Sequential Models

This chapter introduces a non-interleaving (also called true concurrency) model which interprets concurrency as independence between actions rather than as interleavings. As in the previous chapter, we present the system model (Petri nets) and its branching semantics by means of behavior models (occurrence nets and event structures) together with the different notions of observations for this semantics and the **co-ioco** conformance relation that allows to test for independent between actions.

## 3.1 Petri Nets

A net consists of two disjoint sets $P$ and $T$ representing respectively places and transitions together with a set $F$ of flow arcs. The notion of "state" of the system in a net is captured by its markings. A marking is a multiset $M$ of places, i.e. a map $M : P \to \mathbb{N}$. In this thesis, we focus on the so-called safe nets where markings are sets, i.e. $M(p) \in \{0, 1\}$ for all $p \in P$. A Petri net (PN) is a net together with an initial marking and a function that labels its transitions over an alphabet $L$ of observable actions. As in the case of LTS, we differentiate between inputs proposed by the environment and outputs produced by the system ($L = \mathcal{I}n \uplus \mathcal{O}ut$) and internal actions labeled by $\tau$.

**Definition** 3.1 (Petri Nets). *A PN is a tuple $\mathcal{N} = (P, T, F, \lambda, M_0)$ where*

- *$P \neq \emptyset$ is a set of places;*
- *$T \neq \emptyset$ is a set of transitions such that $P \cap T = \emptyset$;*
- *$F \subseteq (P \times T) \cup (T \times P)$ is a set of flow arcs;*
- *$\lambda : T \to (L \cup \{\tau\})$ is a labeling mapping; and*
- *$M_0 \subseteq P$ is an initial marking.*

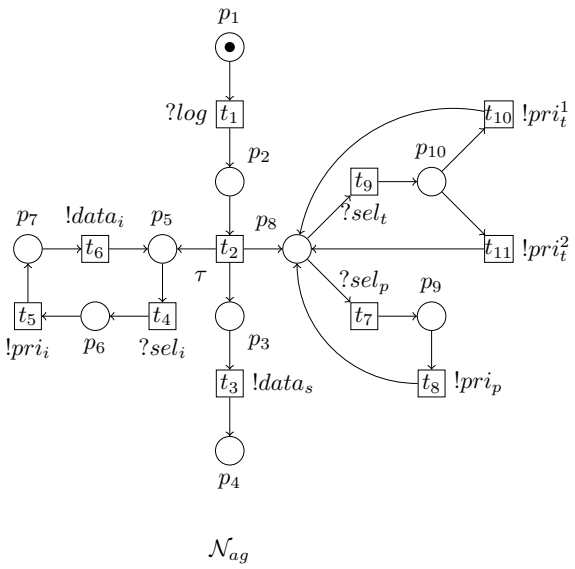



Figure 3.1: A Petri net representing the behavior of a travel agency.

Denote by $T^{\mathcal{I}n}, T^{\mathcal{O}ut}$ and $T^{\tau}$ the input, output and internal transition sets, respectively; that is, $T^{\mathcal{I}n} \triangleq \lambda^{-1}(\mathcal{I}n), T^{\mathcal{O}ut} \triangleq \lambda^{-1}(\mathcal{O}ut)$ and $T^{\tau} \triangleq \lambda^{-1}(\tau)$. Elements of $P \cup T$ are called the nodes of $\mathcal{N}$. For a transition $t \in T$, we call ${}^\bullet t = \{p \mid (p,t) \in F\}$ the preset of $t$, and $t^\bullet = \{p \mid (t,p) \in F\}$ the postset of $t$. These notions can be extended to sets of transitions. In figures, we represent as usual places by empty circles, transitions by squares, $F$ by arrows, and the marking of a place $p$ by black tokens in $p$.

A transition $t$ is enabled in marking $M$, written $M \xrightarrow{t}$, if ${}^\bullet t \subseteq M$. This enabled transition can fire, resulting in a new marking $M' = (M \backslash {}^\bullet t) \cup t^\bullet$. This firing relation is denoted by $M \xrightarrow{t} M'$. A marking $M$ is reachable from $M_0$ if there exists a firing sequence, i.e. transitions, $t_0 \dots t_n$ such that $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \dots \xrightarrow{t_n} M$. The set of reachable markings from $M_0$ is denoted by $\mathcal{R}(\mathcal{N})$.

A Petri net $\mathcal{N}$ is called deterministically labeled if and only if no two transitions with the same label are simultaneously enabled, i.e. for all $t_1 \neq t_2 \in T$ and $M \in \mathcal{R}(\mathcal{N})$ we have $M \xrightarrow{t_1}$ and $M \xrightarrow{t_2}$ imply $\lambda(t_1) \neq \lambda(t_2)$. Deterministic labeling ensures that the system behavior is locally discernible through labels.

Example 3.1 (Petri Nets). *Figure 3.1 shows a PN modeling the behavior of a travel agency. After the user has logged in ($t_1$), the system internally enables (by firing $t_2$) three concurrent behaviors: (i) some data is produced by the server ($t_3$); (ii) the user can choose an insurance ($t_4$); (iii) the user can choose some tickets ($t_7$ or $t_9$). The selection of the insurance is followed by outputs with its price ($t_5$) and some data ($t_6$). If the user selects a plane ticket, its price is shown ($t_8$) and if he selects a train ticket, the system displays one out of two*

*possible prices ($t_{10}$ or $t_{11}$).*

The firing relation allows to represent the dynamic behavior of a net, relating states of the system (markings) and the actions the system can perform in those states (the enabled transitions). However, firing sequences only allow to represent the sequential behavior of a net. We will use partial orders semantics given by a net unfolding which highlights the choices of the system and therefore, we categorize Petri nets as system/non-interleaving/branching-time models.

## 3.2 Partial Order Semantics for PNs

Partial order semantics of a Petri net is given by its unfolding, an acyclic (and usually infinite) structure that highlights the branching of the process. Unfoldings are usually represented by a subclass of Petri nets called occurrence nets. Occurrence nets are isomorphic to event structures [NPW81] and therefore one can easily forget about places of the net by adding information about the branching. Most of the notions presented in this chapter are explained in terms of event structures since they facilitate the presentation. However, for some technical notions in Chapter 5 and Chapter 6, we will use the net representation. We present in this section both formalisms.

The execution for this semantics are not sequences but partial orders, in which concurrency is represented by absence of precedence, i.e. independence.

### 3.2.1 Occurrence Nets and Unfoldings

Occurrence nets can be seen as infinite Petri nets with a special acyclic structure that highlights conflict between transitions that compete for resources. Places and transitions of an occurrence net are usually called conditions and events and denoted by $B$ and $E$. Occurrence nets generalize reachability trees allowing actions to be concurrent. Formally, let $N = (P, T, F)$ be a net, $<$ the transitive closure of $F$, and $\leq$ the reflexive closure of $<$. We say that transitions $t_1$ and $t_2$ are in structural conflict, written $t_1 \#^{\omega} t_2$, if and only if $t_1 \neq t_2$ and ${}^\bullet t_1 \cap {}^\bullet t_2 \neq \emptyset$. Conflict is inherited along $<$, that is, the conflict relation $\#$ is given by $a \# b \Leftrightarrow \exists t_a, t_b \in T : t_a \#^{\omega} t_b \wedge t_a \leq a \wedge t_b \leq b$. Finally, the concurrency relation **co** holds between nodes $a, b \in P \cup T$ that are neither ordered nor in conflict, i.e. $a$ **co** $b \Leftrightarrow \neg(a \leq b) \wedge \neg(a \# b) \wedge \neg(b \leq a)$.

**Definition** 3.2 (Occurrence Nets). *A net $ON = (B, E, F)$ is an occurrence net iff*

1. *$\leq$ is a partial order;*
2. *for all $b \in B$, $|{}^\bullet b| \in \{0, 1\}$;*
3. *for all $x \in B \cup E$, the set $\{y \in E \mid y \leq x\}$ is finite;*
4. *there is not self-conflict, i.e. there is no $x \in B \cup E$ such that $x \# x$;*

*5.* $\perp \in E$ *is the only* $\leq$*-minimal node (event* $\perp$ *creates the initial conditions)*

Call the elements of $E$ events, those of $B$ conditions. A set of conditions is a co-set if its elements are pairwise in **co** relation. A maximal co-set with respect to set inclusion is called a cut. Occurrence nets are the mathematical form of the partial order unfolding semantics of a Petri net [ERV02].

**Definition** 3.3 (Branching Processes). *A branching process of a Petri net* $\mathcal{N} = (P, T, F, \lambda, M_0)$ *is given by a pair* $\beta = (ON, \varphi)$*, where* $ON = (B, E, F)$ *is an occurrence net, and* $\varphi : B \cup E \to P \cup T$ *is such that:*

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

The unique (up to isomorphism) maximal branching process $\mathcal{U} = (ON, \varphi)$ of $\mathcal{N}$ is called the unfolding of $\mathcal{N}$. Every cut $C$ of the unfolding represents a marking in the original net, i.e. $\varphi(C) \in \mathcal{R}(\mathcal{N})$.

We present the algorithm to construct the unfolding of a net [ERV02]. A branching process of a net $\mathcal{N}$ is represented as a set of nodes. A node is either a condition or an event. A condition is a pair $(p, e)$, where $p$ is a place of $\mathcal{N}$ and $e$ is its preset. An event is a pair $(t, C)$, where $t$ is either a transition in $\mathcal{N}$ or $\perp$, and $C$ is its preset. The possible extensions of a branching process $\beta$ are the pairs $(t, C)$ where the elements of $C$ form a co-set, $t$ is such that $\varphi(C) = {}^\bullet t$ and $\beta$ contains no event $e$ satisfying $\varphi(e) = t$ and ${}^\bullet e = C$. We denote the set of possible extensions of $\beta$ by $PE(\beta)$.

The algorithm starts with a branching process containing the $\perp$ event and conditions corresponding to the initial marking. New events from $PE(\beta)$ are added one at a time with its corresponding postset.

**Algorithm 2** Petri nets unfolding

**Require:** a Petri net $\mathcal{N} = (P, T, F, \lambda, M_0)$, where $M_0 = \{p_1, \ldots, p_k\}$
**Ensure:** the unfolding of $\mathcal{N}$

```
U := {(⊥, ∅), (p_1, ⊥), ..., (p_k, ⊥)}
pe := PE(U)
while pe ≠ ∅ do
    add to U an event e = (t, C) of pe and a condition (p, e) for every place p in t•
    pe := PE(U)
return U
```

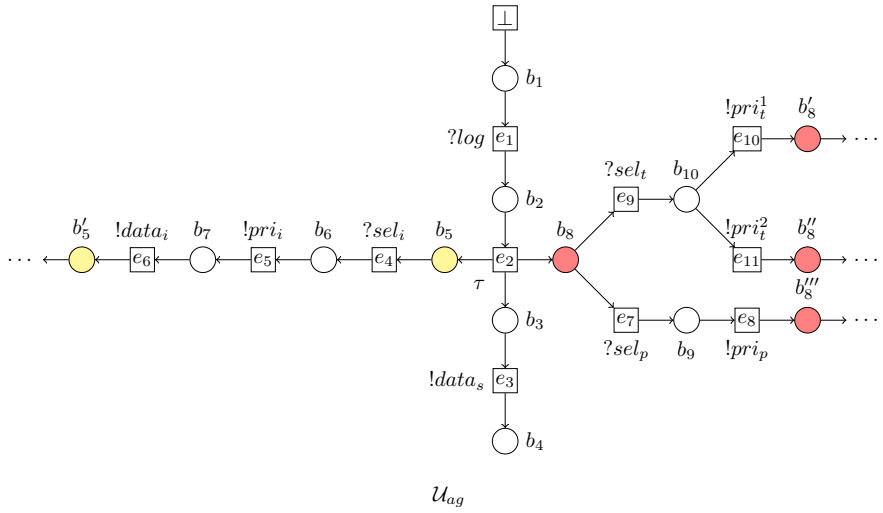



Figure 3.2: Initial part of the unfolding of $\mathcal{N}_{ag}$.

Occurrence nets are a subclass of Petri nets and thus they can be classified as system models. However, as in the case of RTs, we classify them as behavior models as events can occur only once and therefore their execution identify the current state of the system, i.e. the current cut of the unfolding or the marking of the net.

Example 3.2 (Unfoldings). *Figure 3.2 shows the initial part of the unfolding $\mathcal{U}_{ag}$ of the net $\mathcal{N}_{ag}$ in Figure 3.1 obtained by Algorithm 2. Colors show that conditions represent different instances of the same place in the original net, for example, $\varphi(b_8) = \varphi(b'_8) = \varphi(b''_8) = \varphi(b'''_8) = p_8$. As can be observed in the figure, the unfolding of every net containing cycles is infinite.*

### 3.2.2 Event Structures

Occurrence nets give rise to event structures [NPW81] where the information provided by conditions is replaced by a conflict relation.

**Definition** 3.4 (Event Structures). *An Event Structure (ES) over an alphabet $L = \mathcal{I}n \uplus \mathcal{O}ut$ 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. $\forall e, e', e'' \in E : e\#e' \wedge e' \leq e'' \Rightarrow e\#e''$;*

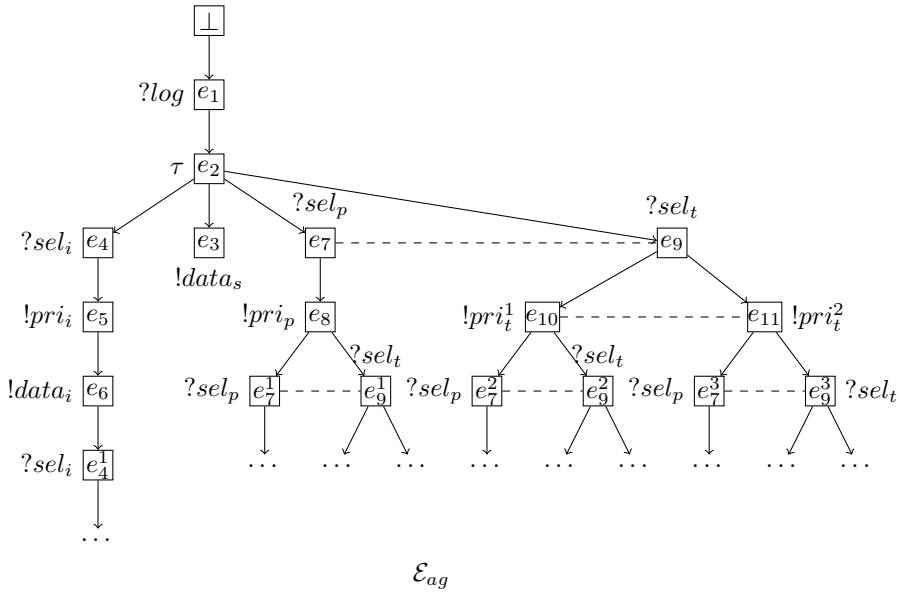



Figure 3.3: Initial part of the unfolding of net from Figure 3.1 given as an ES.

- $\lambda : E \to L \cup \{\tau\}$ *is a labeling mapping;*
- *there exists an unique* $\leq$*-minimal event* $\perp$ *labeled by* $\tau$*.*

Conflicts which cannot be derived from the conflict heredity property are called immediate (or reduced) conflicts and denoted by $e \#^r e'$. In figures, events are represented by squares, causality by arrows and immediate conflict by dashed lines. The past of an event $e$ is defined as $[e] \triangleq \{e' \in E \mid e' \leq e\}$. The sets of inputs, outputs and internal events are denoted respectively by $E^{\mathcal{I}n} \triangleq \{e \in E \mid \lambda(e) \in \mathcal{I}n\}, E^{\mathcal{O}ut} \triangleq \{e \in E \mid \lambda(e) \in \mathcal{O}ut\}$ and $E^{\tau} \triangleq \{e \in E \mid \lambda(e) = \tau\}$. Events which are neither related by causality nor by conflict, are called concurrent, i.e. $e$ **co** $e' \Leftrightarrow \neg(e \leq e') \wedge \neg(e \# e') \wedge \neg(e' \leq e)$.

Example 3.3 (Event Structures). *Figure 3.3 shows the initial part of the unfolding of* $\mathcal{N}_{ag}$ *given as an ES. This ES can easily be obtained from the occurrence net* $\mathcal{U}_{ag}$ *replacing conditions by conflicts.*

An event structure is a behavior/non-interleaving/branching-time model, where the 'state' of the system is represented by the events that have occurred in the computation. As causality represents precedence, such computation must be causally closed. In addition, as conflict represents fight for a resource (this can be observed explicitly in the occurrence net), the computation must be in addition conflict-free. The state of a system in an ES is captured by the notion of configuration.

**Definition** 3.5 (Configurations). *A configuration of an ES is a non-empty set $C \subseteq E$ such that*

- *$C$ is causally closed, i.e. $e \in C$ implies $[e] \subseteq C$, and*
- *$C$ is conflict-free, i.e. $e \in C$ and $e\#e'$ imply $e' \notin C$.*

*For technical convenience, we define all configurations to be non-empty; the initial configuration of $\mathcal{E}$, containing only $\bot$ and also denoted by $\bot$, 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 3.4 (Configurations). *Consider system $\mathcal{E}_{ag}$ from Figure 3.3. The set $\{\bot, e_2, e_4\}$ does not form a configuration as $e_4$ causally depends on $e_1$ which is not part of the set, i.e. the user cannot select an insurance without logging in. The set $\{\bot, e_1, e_2, e_7, e_9\}$ is not conflict-free and thus is not a configuration either, i.e. the user cannot select simultaneously a train and a place ticket. However, the user can login (after which the server can fire the internal action represented by $e_2$), select an insurance and obtain a price and some data, i.e. $C = \{\bot, e_1, e_2, e_4, e_5, e_6\} \in \mathcal{C}(\mathcal{E}_{ag})$. Configurations allow to represent concurrent behaviors, i.e. together with the selection of the insurance, some data can be produced concurrently and thus $C \cup \{e_3\}$ is also a configuration.*

We define configurations in terms of the events of an event structure, however they can be defined directly over occurrence nets. Every configuration $C$ in an occurrence net generates a cut $C^\bullet$ which, as we saw, is related with a reachable marking of the original net. We relate a configuration $C$ and the reachable marking it generates by $Mark(C) = \varphi(C^\bullet)$.

### 3.2.3 Partial Order Executions

The definition of the notion of execution for an event structure is not straightforward since it relies on the chosen semantics for concurrency [ADF86]. For partial order semantics, we use labeled partial orders to keep concurrency explicit in the executions.

**Definition** 3.6 (Labeled Partial Orders). *A Labeled Partial Order (LPO) 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 $\mathcal{LPO}(L)$.*

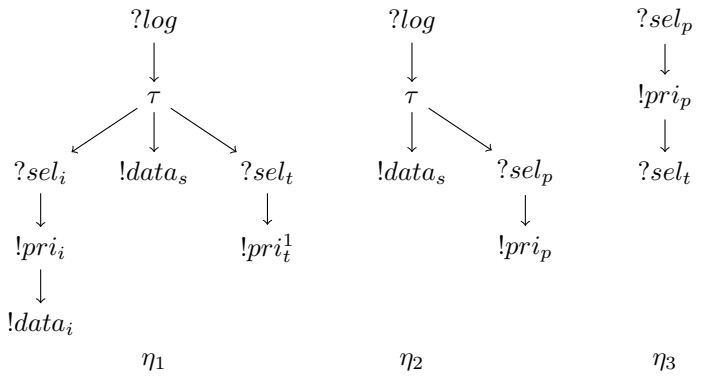



Figure 3.4: Partial order executions.

Example 3.5 (Labeled Partial Orders). *Consider the LPOs from Figure 3.4: $\eta_1$ represents the behavior where after logging in, some internal action is possible, followed by some data which is produced by the server. Concurrently to this, the user selects an insurance and a train ticket with the corresponding production of outputs. In $\eta_2$, some data is produced after logging in and an internal action of the server. Concurrently to the production of data, the user selects a train ticket and its price is produced. In $\eta_3$, the user selects a plane ticket, obtains its price and then selects a train ticket.*

As we can only observe the ordering between the labels and not between the events, we should consider different partial orders respecting this order as equivalent. An isomorphism between partial orders is a bijective function that preserves ordering and labeling.

**Definition** 3.7 (Isomorphisms). *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** 3.8 (Pomsets). *A partially ordered multiset (pomset) is the isomorphic class of some LPO. Any such class is represented by one of its objects. We denote the class of all pomsets by $\mathcal{PS}(L)$.*

As explained above, an execution of an event structure can be represented by a pomset, leading to the following notion of partial order executions.

**Definition** 3.9 (Partial Order Executions). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ with $C, C' \in \mathcal{C}(\mathcal{E})$ and $\eta \in \mathcal{PS}(L)$, we define*

$$\begin{array}{lll} C \xrightarrow{\eta} C' & \triangleq & \exists lpo = (E_\eta, \leq_\eta, \lambda_\eta) \in \eta : C' = C \uplus E_\eta, \\ & & \leq \cap (E_\eta \times E_\eta) = \leq_\eta \text{ and } \lambda_{|E_\eta} = \lambda_\eta \\ C \xrightarrow{\eta} & \triangleq & \exists C' : C \xrightarrow{\eta} C' \end{array}$$

*We say that $\eta$ is a partial order execution of $C$ if $C \xrightarrow{\eta}$.*

As in the case of RTs, we will identify an ES with its initial configuration, e.g. we equally use $\mathcal{E} \xrightarrow{\eta}$ and $\bot \xrightarrow{\eta}$.

Example 3.6 (Partial Order Executions). *Consider the LPOs from Figure 3.4 and system $\mathcal{E}_{ag}$ in Figure 3.3. The labels from $\eta_1$ preserve the causality of the events in $\mathcal{E}_{ag}$ with the same labels, thus $\mathcal{E}_{ag} \xrightarrow{\eta_1} \{\bot, e_1, e_2, e_3, e_4, e_5, e_6, e_9, e_{10}\}$. The LPO $\eta_2$ also respects the causality and then $\mathcal{E}_{ag} \xrightarrow{\eta_2} \{\bot, e_1, e_2, e_3, e_7, e_8\}$. The notion of partial order execution is not restricted just to the initial configuration of the system. If the user has logged in and the system performs an action internally, i.e. it fires $e_2$, the system is currently in configuration $\{\bot, e_1, e_2\}$ from which the behavior $\eta_3$ is possible. This can be formalized as $\{\bot, e_1, e_2\} \xrightarrow{\eta_3} \{\bot, e_1, e_2, e_7, e_8, e_9^1\}$.*

Deterministically labeled nets unfold into ES where executions uniquely define the reached configuration. Such kind of ES are called deterministic.

**Definition** 3.10 (Determinism). *Let $\mathcal{E} \in \mathcal{ES}(L), C, C', C'' \in \mathcal{C}(\mathcal{E})$ and $\eta \in \mathcal{PS}(L)$, $\mathcal{E}$ is deterministic iff $C \xrightarrow{\eta} C'$ and $C \xrightarrow{\eta} C''$ imply $C' = C''$.*

## 3.3 Observing PNs

In this section, we define the notions of traces, outputs, quiescence and refusals in the context of event structures. As these definition depend on configurations, the same definitions can be used for occurrence nets.

### 3.3.1 Traces

The observable behavior of a system can be captured by abstracting the internal actions from its executions (which are pomset in this setting).

**Definition** 3.11 ($\tau$-abstractions). *Let $\eta, \omega \in \mathcal{PS}(L)$, we say that $abs(\eta) = \omega$ iff there exist $lpo_\eta = (E_\eta, \leq_\eta, \lambda_\eta) \in \eta$ and $lpo_\omega = (E_\omega, \leq_\omega, \lambda_\omega) \in \omega$ such that*

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

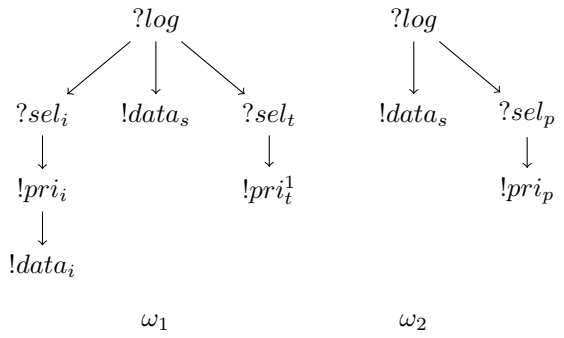



Figure 3.5: $\tau$-abstraction of LPOs.

An observation from a given configuration is the $\tau$-abstraction of one of its partial order executions.

**Definition** 3.12 (Observations). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ with $C, C' \in \mathcal{C}(\mathcal{E})$ and $\omega \in \mathcal{PS}(L)$, we define*

$$\begin{array}{lcl} C \overset{\omega}{\Longrightarrow} C' & \triangleq & \exists \eta : C \xrightarrow{\eta} C' \text{ and } abs(\eta) = \omega \\ C \overset{\omega}{\Longrightarrow} & \triangleq & \exists C' : C \overset{\omega}{\Longrightarrow} C' \end{array}$$

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

Example 3.7 (Observations). *Consider the LPOs from Figure 3.4 and Figure 3.5. Clearly $abs(\eta_1) = \omega_1$ and $abs(\eta_2) = \omega_2$ while $abs(\eta_3) = \eta_3$. It is shown in Example 3.6 that*

$$\begin{array}{rcl} \mathcal{E}_{ag} & \xrightarrow{\eta_1} & \{\bot, e_1, e_2, e_3, e_4, e_5, e_6, e_9, e_{10}\} \\ \mathcal{E}_{ag} & \xrightarrow{\eta_2} & \{\bot, e_1, e_2, e_3, e_7, e_8\} \\ \{\bot, e_1, e_2\} & \xrightarrow{\eta_3} & \{\bot, e_1, e_2, e_7, e_8, e_9^1\} \end{array}$$

*therefore we can conclude that*

$$\begin{array}{rcl} \mathcal{E}_{ag} & \overset{\omega_1}{\Longrightarrow} & \{\bot, e_1, e_2, e_3, e_4, e_5, e_6, e_9, e_{10}\} \\ \mathcal{E}_{ag} & \overset{\omega_2}{\Longrightarrow} & \{\bot, e_1, e_2, e_3, e_7, e_8\} \\ \{\bot, e_1, e_2\} & \overset{\eta_3}{\Longrightarrow} & \{\bot, e_1, e_2, e_7, e_8, e_9^1\} \end{array}$$

We can now define the notions of traces and reached configurations from another given configuration by an observation. Our notion of trace is similar to the one of Ulrich and König [UK97], where a trace is considered as a sequence of partial orders. The reached 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 [GL03].

**Definition** 3.13 (Traces and Reached Configurations). *Let $\mathcal{E} \in \mathcal{ES}(L)$ with $C, C' \in \mathcal{C}(\mathcal{E})$ and $\omega \in \mathcal{PS}(L)$, we define*

$$\begin{array}{rcl} \mathit{traces}(\mathcal{E}) & \triangleq & \{\omega \in \mathcal{PS}(L) \mid \bot \stackrel{\omega}{\Longrightarrow}\} \\ C \ \textbf{\textit{after}}\ \omega & \triangleq & \{C' \mid C \stackrel{\omega}{\Longrightarrow} C'\} \end{array}$$

Example 3.8 (Traces and Reached Configurations). *It is shown in Example 3.7 that both $\omega_1$ and $\omega_2$ are possible observations from the initial configuration of $\mathcal{E}_{ag}$, thus we have $\omega_1, \omega_2 \in \mathit{traces}(\mathcal{E}_{ag})$. In addition, it is easy to see that $?log$ is observable from the initial configuration, i.e. $\mathcal{E}_{ag} \stackrel{?log}{\Longrightarrow}$. However, as internal actions are not observable, it is not possible to detect if $e_2$ occurred or not, this implies that the reached configuration after logging in is not unique, i.e. $(\mathcal{E}_{ag}\ \textbf{\textit{after}}\ ?log) = \{\{e_1\}, \{e_1, e_2\}\}$.*

As it is shown in the example above, our definition of **after** is general enough to handle nondeterminism in the computation. However, in our testing framework, we will only consider deterministic ES where ambiguity between the reached configuration is only due to internal actions.

### 3.3.2 Quiescence and Produced Outputs

The notion of quiescence in true concurrency models is similar to the one in interleaving models, i.e. a quiescent configuration is such that it can not be extended by output events.

**Definition** 3.14 (Quiescence). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$, a configuration $C \in \mathcal{C}(\mathcal{E})$ is quiescent iff for every $e \in E$ such that $C \cup \{e\} \in \mathcal{C}(\mathcal{E})$ we have $e \notin E^{\mathcal{O}ut}$.*

As explained in the previous chapter, the observation of quiescence is usually made explicit in LTSs by adding self loops labeled by a $\delta$ action on quiescent states. The same can be done for a Petri net: for every marking $M$ that only enables input transitions, we add a transition $t_\delta$ with ${}^\bullet t_\delta = t_\delta{}^\bullet = M$ and $\lambda(t_\delta) = \delta$. In the unfolding of such net, a quiescent marking is represented by a quiescent configuration $C$ such that $C \stackrel{\delta}{\Longrightarrow}$. We avoid adding those transitions to the net and just assume that for every quiescent configuration $C$ of the unfolding we have $C \stackrel{\delta}{\Longrightarrow}$.

In the LTS framework, the produced outputs of the systems are single elements of the alphabet of outputs rather than sequences of them [Tre96a]. Consider a system $\mathcal{S}$ that produced $!a$ followed by $!b$ after $\sigma$, then $\mathrm{out}(\mathcal{S}\ \textbf{after}\ \sigma) = \{!a\}$ and $\mathrm{out}(\mathcal{S}\ \textbf{after}\ (\sigma{\cdot}!a)) = \{!b\}$ rather than $\mathrm{out}(\mathcal{S}\ \textbf{after}\ \sigma) = \{!a{\cdot}!b\}$. 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 $!o_1$ and $!o_2$ and an input $?i$ depending on both outputs. Clearly, an implementation that accepts $?i$ before $!o_2$ should not be considered as correct, but if $?i$ is sent too

early to the system, we may not know if the occurrence of $!o_2$ depends or not on $?i$.

In order to compute the set of possible outputs from a given configuration and detect extra dependancies, from every configuration of the system, a quiescent configuration must be reached after some observation. For this reason, we make the following assumption.

**Assumption** 1. *We consider systems that cannot diverge by infinitely many occurrences of internal or output actions, i.e.* $\forall M \in \mathcal{R}(\mathcal{N}), \sigma \in (\mathcal{O}ut \cup \{\tau\})^* : M \xrightarrow{\sigma} M$ *implies* $|\sigma| < \infty$.

The assumption above implies that for every configuration, there exists a finite partial order execution leading to a quiescent configuration. We define the expected outputs from a configuration as the pomset of outputs leading to a quiescent configuration, or $\delta$ if the configuration is already quiescent. This notion can be extended to a set of configurations.

**Definition** 3.15 (Produced Outputs). *Let* $\mathcal{E} \in \mathcal{ES}(L), C \in \mathcal{C}(\mathcal{E})$ *and* $S \subseteq \mathcal{C}(\mathcal{E})$, *we define*

$$\begin{array}{lcl} out(C) & \triangleq & \{!\omega \in \mathcal{PS}(\mathcal{O}ut) \mid C \stackrel{!\omega}{\Longrightarrow} C' \wedge C' \stackrel{\delta}{\Longrightarrow}\} \cup \{\delta \mid C \stackrel{\delta}{\Longrightarrow}\} \\ out(S) & \triangleq & \bigcup\limits_{C \in S} out(C) \end{array}$$

Example 3.9 (Produced Outputs). *Consider the configuration reached by* $\mathcal{E}_{ag}$ *after the user logged in, selected an insurance and the server produced its data, i.e.* $(\mathcal{E}_{ag}$ ***after*** $?log \cdot (!data_s$ **co** $?sel_i)) = \{\bot, e_1, e_2, e_3, e_4\}$. *From this configuration, the system produces the price of the insurance followed by its data, i.e.* $out(\mathcal{E}_{ag}$ ***after*** $?log \cdot (!data_s$ **co** $?sel_i)) = \{!pri_i \cdot !data_i\}$. *Now consider the configuration just before the server produced the data, i.e.* $(\mathcal{E}_{ag}$ ***after*** $?log \cdot ?sel_i) = \{\bot, e_1, e_2, e_4\}$. *This configuration also enables the outputs corresponding to the insurance price and data, but the configuration the system reaches after they are produced, i.e.* $(\mathcal{E}_{ag}$ ***after*** $?log \cdot ?sel_i \cdot !pri_i \cdot !data_i) = \{e_1, e_2, e_4, e_5, e_6\}$, *is not quiescent as it still enables the production of the server data. We can conclude* $!pri_i \cdot !data_i \notin out(\mathcal{E}_{ag}$ ***after*** $?log \cdot ?sel_i)$. *The output produced from this configuration considers concurrently the outputs of the insurance and the data produced by the server, i.e.* $out(\mathcal{E}_{ag}$ ***after*** $?log \cdot ?sel_i) = \{(!pri_i \cdot !data_i)$ **co** $!data_s\}$. *The set of produced outputs is not necessarily a singleton. After logging in, producing some data and selecting a train ticket, the system produces one out of two possible outputs, i.e.* $out(\mathcal{E}_{ag}$ ***after*** $?log \cdot (!data_s$ **co** $?sel_t)) = \{!pri_t^1, !pri_t^2\}$.

### 3.3.3 Possible Inputs

Possible inputs from a given configuration are pomsets composed of input actions that are enabled. Different to what happens with outputs, where we consider all the outputs leading to a quiescent configuration, for inputs, we consider every possible pomsets, even those representing partial behavior of other possible inputs.

**Definition** 3.16 (Possible Inputs). *Let $\mathcal{E} \in \mathcal{ES}(L), C \in \mathcal{C}(\mathcal{E})$ and $S \subseteq \mathcal{C}(\mathcal{E})$, we define*

$$\begin{array}{rcl} poss(C) & \triangleq & \{?\omega \in \mathcal{PS}(\mathcal{I}n) \mid C \stackrel{?\omega}{\Longrightarrow}\} \\ poss(\emptyset) & \triangleq & \mathcal{PS}(\mathcal{I}n) \\ poss(S) & \triangleq & \bigcup_{C \in S} poss(C) \end{array}$$

Example 3.10 (Possible Inputs). *From the initial configuration of $\mathcal{E}_{ag}$, the user can just log in, i.e. $poss(\mathcal{E}_{ag}) = \{?log\}$, however as soon as he logged in, the insurance selection and the choices between tickets become available either as individual actions or concurrent ones, i.e. $poss(\mathcal{E}_{ag}$ **after** $?log) = \{?sel_i, ?sel_t, ?sel_p, ?sel_i$ **co** $?sel_t, ?sel_i$ **co** $?sel_p\}$. This example shows how partial behaviors are also considered: we do not only consider $?sel_i$ **co** $?sel_t$ as a possible input, but also $?sel_i$ and $?sel_t$.*

## 3.4 The co-ioco Conformance Relation

This section presents the testing hypotheses and conformance relation for conformance testing of Petri nets.

**Testing Hypotheses.** We assume that the specification of the system under test is given as a deterministically labeled Petri net $\mathcal{N} = (P, T, F, \lambda, M_0)$ over alphabet $L = \mathcal{I}n \uplus \mathcal{O}ut$ of input and output labels. To be able to test an implementation against such a specification, we make the usual testing assumption that the behavior of the SUT itself can be modeled by a Petri net over the same alphabet of labels. We also assume as that the specification does not contain cycles of silent or outputs actions (Assumption 1), so that the number of expected outputs after a given trace is finite and that we can observe whenever an input is refused.

The **co-ioco** conformance relation compares outputs and quiescence of the implementation w.r.t those of the specification after an experiment extracted from the specification. However, in this setting, traces and outputs are considered under partial order semantics, i.e. executions and outputs are pomsets where actions specified as concurrent need to be implemented as concurrent. In addition, we drop the input-enabledness assumption on the implementation and consider refusals.

**Definition** 3.17 (**co-ioco**). *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and an implementation of the system, then*

$$\begin{array}{rcl} \mathcal{I} \textbf{ co-ioco } \mathcal{S} & \Leftrightarrow & \forall \omega \in traces(\mathcal{S}) : \\ & & poss(\mathcal{S} \textbf{ after } \omega) \subseteq poss(\mathcal{I} \textbf{ after } \omega) \\ & & out(\mathcal{I} \textbf{ after } \omega) \subseteq out(\mathcal{S} \textbf{ after } \omega) \end{array}$$

Inclusion of possible inputs can be interpreted as "any input refused by the implementation must be refused by the specification".

Example 3.11 (Removed Outputs and Silent Actions). *Figure 3.6 shows a possible implementation of the travel agency specified by* $\mathcal{S}_2 = \mathcal{E}_{ag}$ *in Figure 3.3. Event* $e_2$ *is not implemented, but as this event is not observable from the environment and the transitive causalities (such as* $e_1 \leq e_3$*) are implemented, the absence of* $e_2$ *does not lead to non conformance. In addition, the possibility of producing a second kind of price after the selection of a train is removed, but this is allowed by* **co-ioco**. *The rest of the implementation is isomorphic to* $\mathcal{S}_2$ *and we can conclude* $\mathcal{I}_5$ **co-ioco** $\mathcal{S}_2$.

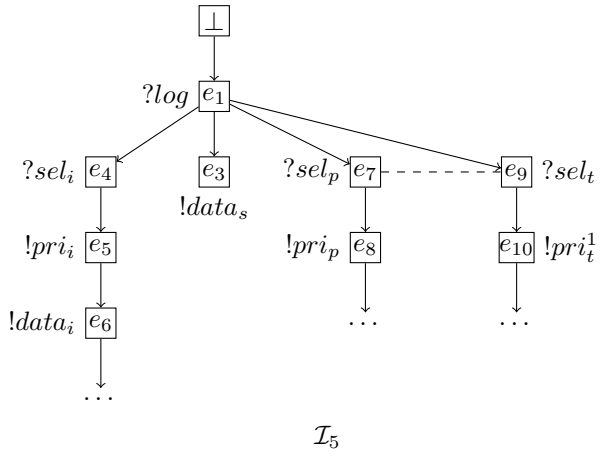



Figure 3.6: A conformant implementation of the travel agency w.r.t **co-ioco**.

Example 3.12 (Removed inputs). *Figure 3.7 shows an implementation that removes the possibility of selecting a train ticket after logging in, i.e.* $?sel_t \notin poss(\mathcal{I}_6$ ***after*** $?log)$. *As this input is possible in* $\mathcal{S}_3$, *we have* $\neg(\mathcal{I}_6$ **co-ioco** $\mathcal{S}_3)$.

Example 3.13 (Extra Causalities). *Figure 3.8 shows three possible implementations of the travel agency. Implementation* $\mathcal{I}_7$ *adds causality* $e_3 \leq e_4$ *and then* $?sel_i \notin poss(\mathcal{I}_7$ ***after*** $?log)$. *Since the selection of the insurance is possible after logging in* $\mathcal{S}_2$, *we have* $\neg(\mathcal{I}_7$ **co-ioco** $\mathcal{S}_2)$. *Implementation* $\mathcal{I}_8$ *also adds order between these events, but in the reverse order, i.e.* $e_4 \leq e_3$. *This extra causality generates unspecified quiescence in the system, i.e.* $out(\mathcal{I}_8$ ***after*** $?log) = \{\delta\}$ *while* $out(\mathcal{S}_2$ ***after*** $?log) = \{!data_s\}$, *thus* $\neg(\mathcal{I}_8$ **co-ioco** $\mathcal{S}_2)$. *System* $\mathcal{I}_9$ *produces the same output actions of the insurance, but in a concurrent way rather than sequentially, i.e.* $out(\mathcal{I}_9$ ***after*** $?log{\cdot}?sel_i) = \{!pri_i$ **co** $!data_i\} \neq \{!pri_i{\cdot}!data_i\} = out(\mathcal{S}_2$ ***after*** $?log{\cdot}?sel_i)$, *thus we have* $\neg(\mathcal{I}_9$ **co-ioco** $\mathcal{S}_2)$.

Example 3.14 (Extra Behaviors). *Implementations in Figure 3.9 allow extra behaviors to those specified:* $\mathcal{I}_{10}$ *gives the user the choice of selecting a boat ticket rather than a train or plane, while* $\mathcal{I}_{11}$ *allows the user, concurrently with the*

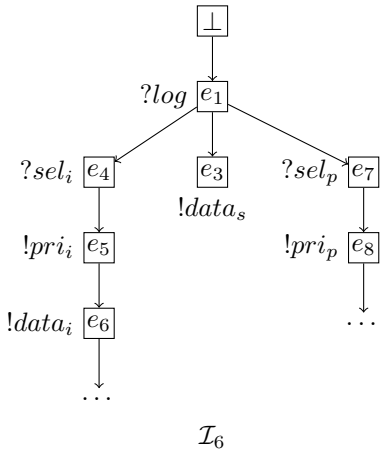



Figure 3.7: Removed inputs.

*other selections, to pick a hotel. Both implementations produced extra outputs ($!pri_b$ and $!pri_h$ respectively), but as these outputs are only produced after underspecified inputs, both implementations are correct, i.e.* $\mathcal{I}_{10}$ **co-ioco** $\mathcal{S}_2$ *and* $\mathcal{I}_{11}$ **co-ioco** $\mathcal{S}_2$.

**Relating co-ioco and ioco:** The following remarks allow to relate the conformance relations of Chapter 2 and the **co-ioco** relation presented in this chapter.

Remark 2. *A labeled transition system can easily be transformed into a Petri net and thus we can easily relate a reachability tree with its corresponding occurrence net or event structure. Given* $(N, L, \leq, n_0) \in \mathcal{RT}(L)$*, its corresponding occurrence net is such that:* $B = N$*;* $M_0 = \{n_0\}$*; if* $\exists (n, a, n') \in \leq$ *then* $\exists e \in E : {}^\bullet e = \{n\}, e^\bullet = \{n'\}$ *and* $\lambda(e) = a$*. It is easy to see that any sequential execution* $n \xrightarrow{\mu_1 \dots \mu_k} n'$ *of the reachability tree leads to a firing sequence* $\{n\} \xrightarrow{t_1 \dots t_k} \{n'\}$ *with* $\lambda(t_i) = \mu_i$ *between the markings of the occurrence net. Every configuration* $C$ *generates a marking of the occurrence net* $C^\bullet$*; then* $\{n\} \xrightarrow{t_1 \dots t_k} \{n'\}$ *generates a partial order execution* $C \xrightarrow{\eta} C'$ *where* $C^\bullet = \{n\}, C'^\bullet = \{n'\}$ *and* $\mu_1 \dots \mu_k$ *is a total order of* $\eta$.

Remark 3. *Whenever a reachability tree specification produces an output, any correct implementation that implements that output, must implement all the path composed by outputs that follow it to avoid extra quiescence. Imagine a specification* $\mathcal{S}$ *such that* $out(\mathcal{S}$ ***after*** $?a) = \{!b\}$ *and* $out(\mathcal{S}$ ***after*** $?a{\cdot}!b) = \{!d\}$*. If an implementation* $\mathcal{I}$ *produces* $!b$ *after* $?a$*, but not* $!d$*, this implementation is non conformant as* $out(\mathcal{I}$ ***after*** $?a{\cdot}!b) = \{\delta\}$*. If we restrict to systems that do*

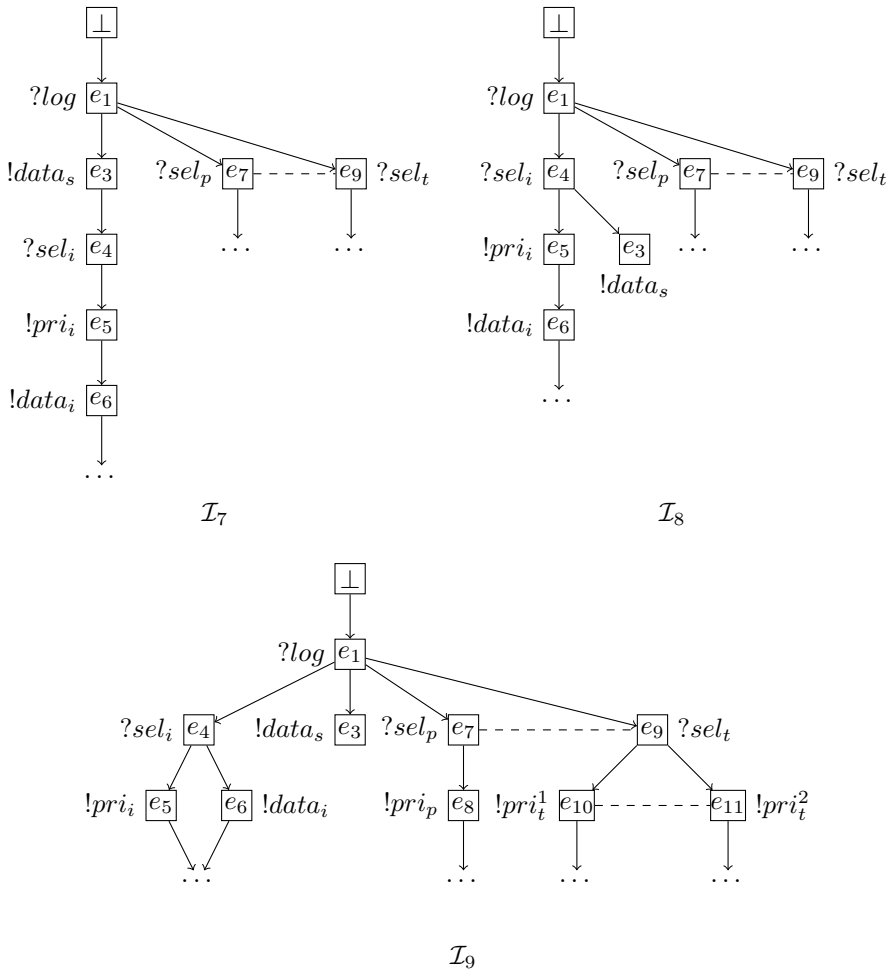



Figure 3.8: Extra causality and concurrency.

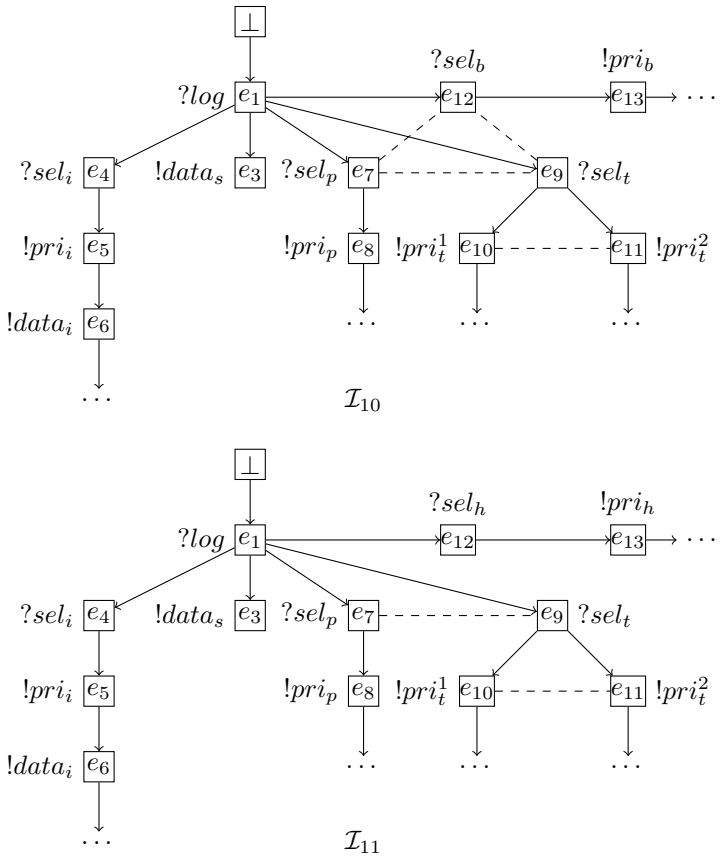



Figure 3.9: Extra conflicting and concurrent inputs.

*not allow cycles of outputs or silent actions, Definition 2.6 can be rewritten as*

$$out(n) \triangleq \{!\sigma \in \mathcal{O}ut^* \mid n \stackrel{!\sigma}{\Longrightarrow} n' \wedge n' \stackrel{\delta}{\Longrightarrow}\} \cup \{\delta \mid n \stackrel{\delta}{\Longrightarrow}\}$$

We can now prove that when there are not cycles of outputs or internal actions in the system, any correct implementation w.r.t **co-ioco** is correct w.r.t $\mathbf{ioco}_2$.

**Theorem** 2. *Let $\mathcal{S}$ and $\mathcal{I}$ be respectively the specification and implementation of a system, then we have*

$$\mathcal{I} \textbf{ co-ioco } \mathcal{S} \textit{ implies } \mathcal{I} \textbf{ ioco}_2 \mathcal{S}$$

*Proof.* Immediate using Remark 2 and Remark 3. □

In addition, if we assume the input-enabledness of the implementation, any correct implementation w.r.t **co-ioco** is correct w.r.t **ioco**.

**Theorem** 3. *Let $\mathcal{S}$ and $\mathcal{I}$ be respectively the specification and an input-enabled implementation of a system, then we have*

$$\mathcal{I} \textbf{ co-ioco } \mathcal{S} \textit{ implies } \mathcal{I} \textbf{ ioco } \mathcal{S}$$

*Proof.* By Theorem 1 and Theorem 2. □

## 3.5 Conclusion

Even if the testing problem for concurrent systems have been widely studied in the past, it was mostly in the context of interleaving semantics. Besides the state space explosion problem that interleavings can generate, we have shown in the introduction that independence between actions can not be tested using this semantics.

This chapter presents the basic notions for the definition of a testing framework where independence between actions plays a central role. The formal model that we use for describing the behavior of the system is (1-safe) Petri nets and their unfoldings. The explicit representation of concurrency avoids the state space problem in the generation of test cases (see Chapter 5) and makes the independence of action observable: in this setting, executions and observations are partial orders where lack of dependence is interpreted as independence.

Based on the notions of (partial order) executions and observations, we defined the **co-ioco** conformance relation where extra or missing causalities can be detected: any system that implements actions that were specified as concurrent or independent by any possible order (interleavings) is considered incorrect. In addition, the proposed conformance relation drops the input-enabledness assumption on the implementation and allows to test for refusals.

Finally, we showed that **co-ioco** is a generalization of **ioco** in the sense that when the system does not contain concurrency and the implementation is input-enabled, both relations coincide.

# 4

# Conformance Testing with Refined Concurrency

Chapter 2 and Chapter 3 present the two standard semantics for concurrency, together with their corresponding notions of executions, observations and conformance relations. However, both semantics may be too strict in some situations where concurrency is implemented either using interleaving or partial order semantics depending on, for example, the architecture of the system. Figure 4.1 presents a specification Spe with two processes $P_1$ and $P_2$ (represented by boxes) and three concurrent actions $a, b, c$. Actions $a$ and $c$ belong to process $P_1$ while $b$ belongs to process $P_2$. In a distributed architecture, when two actions belong to different processes, they are specified as concurrent and therefore should be implemented in different processes. We call this notion strong concurrency and interpret it as independence between actions, meaning that there should not be any kind of causality (drawn by arrows) between these actions. In Spe, actions $a$ and $b$ are strongly concurrent (as they belong to different processes) and are implemented in different processes in both $\text{Impl}_1$ and $\text{Impl}_2$. However, in an early stage of specification, concurrency between events can be used as underspecification or possible refinement. 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 $\text{Impl}_1$; 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 $\text{Impl}_2$. We capture underspecification and refinement with the notion of weak concurrency. As it is the case of local trace languages [KM02], we allow actions to be independent in one situation, while in another, they cannot be performed independently, which means that the same actions can be specified as strongly concurrent in a part of the specification and weakly concurrent in another part of the same specification.

We illustrate the need to make these two notions of concurrency live in the same model by examples coming from the field of micro-controller design and security protocols.

Example 4.1 (Weak Concurrency). *Consider a ParSeq controller [MY10] which*

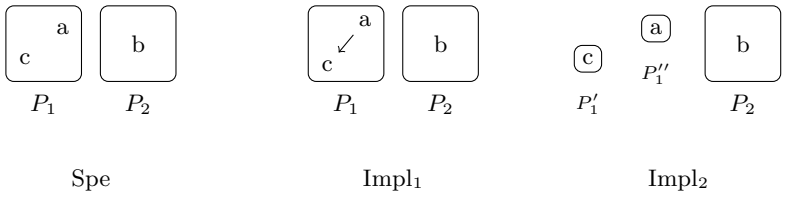



Figure 4.1: A specification of a system and two possible implementations.

*manages two handshakes* $A = (rek_a, ack_a)$ *and* $B = (rek_b, ack_b)$*, and a set of Boolean variables* $x_1, x_2, x_3$ *provided by the environment as shown in Figure 4.2. 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 handshakes are 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* $B$ *are specified as weakly concurrent and their actual order (if any) depends on the values of the variables.*

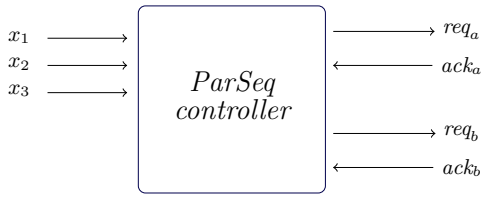



Figure 4.2: ParSeq controller interface.

Example 4.2 (Strong Concurrency). *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 [ACRR10]. Causality captures linkability as two messages coming from the same user need to be causally dependent. However, concurrency interpreted as interleavings cannot be used to model unlinkability because both possible interleavings relate the messages and therefore they reveal the identity of the user. This property needs to be modeled by strong concurrency.*

The notions of weak and strong concurrency were introduced in [PHL14c] directly over event structures. To follow the presentation of the previous two chapters, we introduce a system model whose semantics can be described by event structures with weak and strong concurrency. Conditional partial order graphs can handle concurrency as independence and interleavings. We show how

to unfold those graphs into an event structures that express their semantics in the presence of weak and strong semantics. Different notions of observation and a new conformance relation are presented under this semantics.

## 4.1 Conditional Partial Order Graphs

We are interested in a model that allows to interpret both kinds of concurrency and that represents in an compact way the behavior of a concurrent system. A Conditional Partial Order Graph (CPOG) [MY10] is an structure that allows compact representation of a set of partial orders. CPOGs can be used to represent different behaviors of a concurrent system. This structure consists of a directed graph whose vertices and edges are labeled by Boolean conditions over a set of variables $X$. An opcode is an assignment $(x_1, x_2, \ldots, x_{|X|}) \in \{0,1\}^{|X|}$ of these variables; these assignments should satisfy the restriction function $\rho$ of the graph, i.e. $\rho(x_1, x_2, \ldots, x_{|X|}) = 1$. Vertices of the graph represent actions that the system can perform and edges the dependancies between them.

**Definition** 4.1 (Conditional Partial Order Graphs). *A CPOG is a quintuple $H = (V, A, X, \phi, \rho)$, where*

- *$V$ is a set of vertices,*
- *$A$ is a set of arcs between them,*
- *$X$ is a set of Boolean variables,*
- *function $\phi$ assigns a Boolean condition $\phi(z)$ to every vertex and arc $z \in V \uplus A$ of the graph,*
- *$\rho : 2^{|X|} \to \{0,1\}$ is a restriction function.*

Example 4.3 (Conditional Partial Order Graphs). *Consider a travel agency that allows the user, after logging in, only to choose between selecting an insurance or a plane ticket. If a plane ticket is selected, its price is displayed. However, if an insurance is selected, the agency displays some data and its price, but it can decide to display these outputs either concurrently or in a specific order. Figure 4.3 (below) shows the four possible behaviors of the system and a CPOG (above) representing these behaviors. There are two operational variables $x$ and $y$ and the restriction function is $\rho = 1$, hence, the four opcodes $x, y \in \{0,1\}$ are allowed. Vertices and arcs labeled by 1 are called unconditional (conditions equal to 1 are not depicted in the graph). Vertices and arcs are labeled with predicates $\Phi = \{1, \overline{x} \vee \overline{y}, x \wedge y, x, y\}$ with the following labeling:*

$$
\begin{array}{llllllllll}
\phi_{?log} & = & 1 & \phi_{!pri_p} & = & x \wedge y & \phi_{?sel_i \to !pri_i} & = & \overline{y} \\
\phi_{?sel_i} & = & \overline{x} \vee \overline{y} & \phi_{?log \to ?sel_p} & = & 1 & \phi_{?sel_i \to !data_i} & = & \overline{x} \\
\phi_{!pri_i} & = & \overline{x} \vee \overline{y} & \phi_{?sel_p \to !pri_p} & = & 1 & \phi_{!pri_i \to !data_i} & = & x \\
\phi_{!data_i} & = & \overline{x} \vee \overline{y} & \phi_{?log \to ?sel_i} & = & 1 & \phi_{!data_i \to !pri_i} & = & y \\
\phi_{?sel_p} & = & x \wedge y & & & & & &
\end{array}
$$

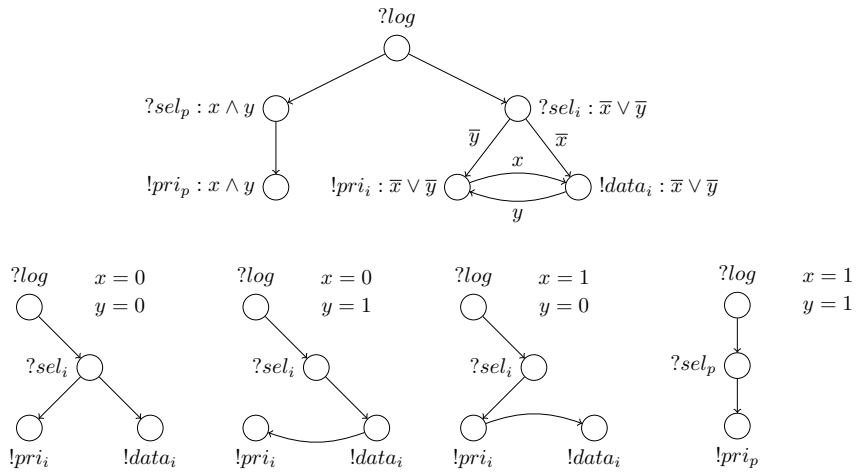



Figure 4.3: Conditional partial order graph and its corresponding set of partial orders.

The purpose of conditions is to 'switch off' some vertices and/or arcs in the graph according to the given opcode. This makes CPOGs capable of containing multiple projections as shown in Figure 4.3. The leftmost projection is obtained by keeping in the graph only those vertices and arcs whose conditions evaluate to Boolean 1 after substitution of the operational variables $x$ and $y$ with Boolean 0. Hence, vertex $?sel_p$ disappears, because its condition evaluates to 0: $\phi_{?sel_p} = x \wedge y = 0$. Arcs $!pri_i \rightarrow !data_i$ and $!data_i \rightarrow !pri_i$ disappear for the same reason. Note also that although the condition on arc $?log \rightarrow ?sel_p$ evaluates to 1 (in fact it is constant 1) the arc is still excluded from the projection because one of the vertices it connects (vertex $?sel_p$) is excluded and an arc cannot appear in a graph without one of its adjacent vertices. Branching of the system is not represented explicitly: different scenarios are captured by different projections, thus CPOGs are categorized as behavior/non-interleaving/linear-time models.

Each projection is treated as a partial order specifying a behavioral scenario of the modeled system. Potentially, a CPOG $H = (V, A, X, \phi, \rho)$ can specify an exponential number of different partial orders on actions $V$ according to $2^{|X|}$ possible opcodes. We will use notation $H_{|\psi}$ to denote a projection of a graph $H$ under opcode $\psi = (x_1, x_2, \ldots x_{|X|})$. A projection $H_{|\psi}$ is called valid iff opcode $\psi$ is allowed by the restriction function, i.e. $\rho(x_1, x_2, \ldots x_{|X|}) = 1$, and the resulting graph is acyclic. The latter requirement guarantees that the graph defines a set of partial orders. A CPOG is well-formed iff every allowed opcode produces a valid projection. The graph in Figure 4.3 is well-formed, because the opcodes satisfy the restriction function and its projections are acyclic. The set of partial order defined by a well-formed graph $H$ is denoted by $P(H)$.

## 4.2 Semantics for Weak and Strong Concurrency

As in the case of LTSs or PNs, CPOGs can be unfolded to express their semantics. We unfold the graph into an event structure that represents its partial order semantics. However, event structures do not directly allow to represent weak and strong concurrency. We split the concurrency relation into two relations (weak and strong concurrency) and define a notion of execution that keeps strong concurrent actions as independent, but that may order some weakly concurrent actions.

### 4.2.1 Unfolding of a CPOG

The partial order semantics of a CPOG can be expressed by an event structure obtained by unfolding the graph (in order to obtain an acyclic structure) and replacing Boolean conditions by conflicts. We start from an empty event structure $(E, \leq, \#, \lambda)$ where $E = \emptyset$ and at each iteration, we compute the set of possible extensions. Unfolding a CPOG is complex problem since deciding which are the possible extensions of a prefix of the unfolding is an NP-hard problem.

**Theorem** 4. *Given a CPOG and a prefix of its unfolding, deciding if an instance of a vertex is a possible extension is NP-hard.*

*Proof.* Consider a CPOG containing vertices $v_1, v_2$ and $v_1 \to v_2$ with conditions $\phi_{v_1}$ and $\phi_{v_2} = \phi_{v_1 \to v_2} = 1$. We need to be able to check that $\phi_{v_1}$ is not constantly equal to 0 or 1. If $\phi_{v_1} = 0$, then the unfolding should only contain $v_2$; if $\phi_{v_1} = 1$, the unfolding should contain $v_1$ and $v_2$ with $v_1 \leq v_2$; otherwise, it has three events $v_1, v_2, v_2'$ with $v_1 \leq v_2$ and $v_1 \# v_2'$ (that is, $v_1$ either happens or not). Obviously, checking if $\phi_{v_1}$ is equal to 0 or 1 is an NP-complete problem. □

To decide if an instance of vertex $x \in V$ is a possible extension, we need to find a set of predecessors events in the ES such that *(i)* the Boolean condition of the vertex is true; *(ii)* the Boolean conditions of the instances of its predecessors and their corresponding arcs are true; *(iii)* if an event is not a predecessor, then either its Boolean condition is false, or a Boolean condition of its corresponding arcs is false; *(iv)* the instance of the vertex is different from any other in the prefix. This is captured by the following formula for each vertex $x \in V$:

$$\phi_x \wedge \Big( \bigwedge_{\substack{e_y \in P \\ y \to x \in A}} \phi_{e_y} \wedge \phi_{y \to x} \Big) \Big( \bigwedge_{\substack{e_y \in E \setminus P \\ y \to x \in A}} \neg \phi_{e_y} \vee \neg \phi_{y \to x} \Big) \Big( \bigwedge_{e_x \in E} \neg \phi_{e_x} \Big) \tag{4.1}$$

Whenever such a combination exists, we add the event to the unfolding, appropriately connecting it to its predecessors and with its corresponding Boolean condition. The unfolding procedure is finished when Equation 4.1 is no longer satisfiable: the method always finishes as the CPOG represents finite scenarios. When there does not exist any possible extension, Boolean conditions are replaced by conflicts in the following way: for every pair of events $e_x, e_y$ with

mutually exclusive conditions, i.e. $\neg\varphi_{e_x} \vee \neg\varphi_{e_y}$, conflict $e_x \# e_y$ is added and their Boolean conditions are removed.

The unfolding method is deterministic: the resulting event structure does not depend on the order in which events are added into the unfolding.

**Proposition** 1. *Let $E$ be the current set of events of the unfolding and $e_a \neq e_b$ two possible extensions, then $e_b$ is a possible extension of $E \cup \{e_a\}$.*

*Proof.* We need to prove that Equation 4.1 is still satisfiable for vertex $b$ when $e_a$ is added to the unfolding: *i)* as $e_b$ is a possible extension from $E$, $\phi_b = 1$ and this is also true from $E \cup \{e_a\}$; *ii-iii)* since $P \subseteq E \subseteq E \cup \{e\}$ the second and third conjunction of Equation 4.1 are still satisfied; *iv)* since $e_a \neq e_b$ and there was not an instance of $b$ in $E$, there is neither in $E \cup \{e_a\}$. □

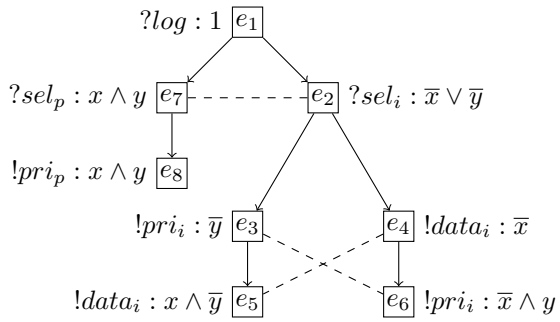



Figure 4.4: Transformation of a CPOG into an ELES.

The unfolding method for CPOG is different from the unfoldings algorithms for PNs. The obtained event structure is conflict-free: possible extensions captured by Equation 4.1 are concurrent and stamped in conflict later on, depending on the boolean variables.

Example 4.4 (CPOG Unfolding). *Consider the CPOG shown in Figure 4.3. The unfolding procedure starts with $E = \emptyset$ and keeps checking vertices of the CPOG for possible extensions (see Figure 4.4). At start, only vertex ?log is a possible extension and event $e_1$ can be added to the unfolding. For other vertices, for example $?sel_i$, the constraint imposed by non-predecessors in Equation 4.1 will include $\neg\phi_{?log \to ?sel_i} = \neg 1 = 0$, hence it is not a possible extension at start. For the same reason vertices $!pri_i, !data_i, ?sel_p, !pri_p$ are not possible extensions when $E = \emptyset$. We proceed by adding event $e_1$ to the unfolding with $\phi(e_1) = 1$. When we recompute the possible extensions, formula Equation 4.1 reduces to $\overline{x} \vee \overline{y}$ and $x \wedge y$ for vertices $?sel_i$ and $?sel_p$, respectively, therefore events $e_2$ and $e_7$ are added with $\phi(e_2) = \overline{x} \vee \overline{y}, \phi(e_7) = x \wedge y$ and $e_1$ as their predecessor. At this point $E = \{e_1, e_2, e_7\}$ and we find that $!pri_i, !data_i$ and $!pri_p$ are possible extensions with events $e_2$ (for $!pri_i$ and $!data_i$) and $e_7$ (for $!pri_p$) as predecessors and conditions $\overline{y}, \overline{x}$ and $x \wedge y$ respectively. Events $e_3, e_4$ and $e_8$ are added*

*and from* $E = \{e_1, e_2, e_3, e_4, e_7, e_8\}$ *we find that* $!pri_i$ *and* $!data_i$ *are possible extensions again. Two new events* $e_5$ *and* $e_6$ *are added with* $e_3$ *and* $e_4$ *as their respective predecessors and* $x \wedge \overline{y}$ *and* $\overline{x} \wedge y$ *as Boolean conditions. Finally, as* $E$ *grows to* $\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$*, Equation 4.1 becomes unsatisfiable and the unfolding procedure is finished. Boolean conditions of events* $e_2$ *and* $e_7$ *are mutually exclusive:* $(x \wedge y) \wedge (\overline{x} \vee \overline{y}) = 0$*, therefore we add immediate conflict* $e_2 \#^r e_7$*. Due to the same reasoning, immediate conflicts* $e_3 \#^r e_6$ *and* $e_4 \#^r e_5$ *are added. Finally, when all Boolean conditions are removed, we obtain an event structure.*

Even if the semantics of a CPOG can be expressed in terms of an event structure using the unfolding algorithm presented above, similarities between different scenarios are not considered. In Figure 4.4, after the system performed actions $?log, ?sel_i$ and $!pri_i$, action $!data_i$ is possible either concurrently with $!pri_i$ or causality depending on it. These two possible continuations are represented by two different events $e_4$ and $e_5$. In order to remove this redundancy and reduce the size of the event structure, we propose to split the **co** relation into two relations: **sco** representing strong concurrency, and **wco** representing weak concurrency, i.e. $\mathbf{co} = \mathbf{sco} \uplus \mathbf{wco}$. This allows to remove events $e_5$ and $e_6$ by setting $e_3$ **wco** $e_4$. The statement $e_3$ **wco** $e_4$ implies that from configuration $\{e_1, e_2\}$, actions $!pri_i$ and $!data_i$ are possible either concurrently or in any order.

### 4.2.2 Relaxed Executions

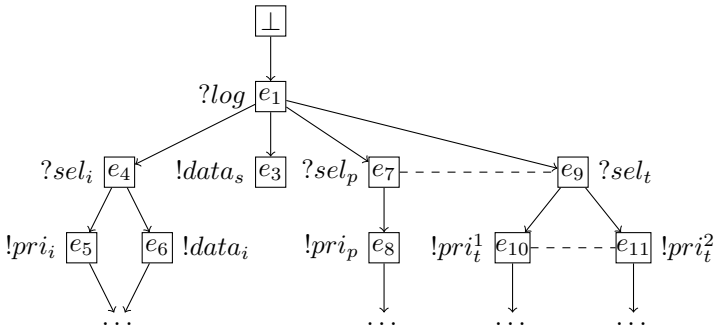
The notions of weak and strong concurrency are defined over the concurrency relation of the event structure, but the same distinction can be made in any concurrency relation, for example the concurrency relation of a partial order. Weak concurrency allows, but does not impose, independence between actions, thus, an execution of the system has to preserve the partial order semantics, up to adding order between weakly concurrent events. On the contrary, strongly concurrent events must remain independent. We capture these notions by pomset refinement.

**Definition** 4.2 (Pomset Refinement). *Let* $\eta_1, \eta_2 \in \mathcal{PS}(L)$*, we say that* $\eta_2$ *refines* $\eta_1$*, denoted by* $\eta_1 \sqsubseteq \eta_2$*, iff there exist* $lpo_1 = (E, \leq_{\eta_1}, \lambda) \in \eta_1$ *and* $lpo_2 = (E, \leq_{\eta_2}, \lambda) \in \eta_2$ *such that*

- $\leq_{\eta_1} \subseteq \leq_{\eta_2}$
- $\boldsymbol{sco}_1 = \boldsymbol{sco}_2$

*In other words,* $\eta_1 \sqsubseteq \eta_2$ *if strong concurrency is preserved, while weakly concurrent events from* $\eta_1$ *may be ordered by* $\leq_{\eta_2}$*.*

Example 4.5 (Pomset Refinement). *Consider* $\mathcal{S}_3$ *in Figure 4.5 as the travel agency specification, where strong concurrency is only between the actions of different suppliers:* $\mathbf{sco} = \{?sel_i, !pri_i, !data_i\} \times \{?sel_t, !pri_t^1, !pri_t^2, ?sel_p, !pri_p\}$*. Weak concurrency is between actions that belong either to the ticket or insurance*




$\mathcal{S}_3$

Figure 4.5: A specification with weak concurrency.

*supplier and the data of the server together with the price and data of the insurance, i.e.* $\mathbf{wco} = \{!data_s\} \times \{?sel_i, !pri_i, !data_i, ?sel_t, !pri_t^1, !pri_t^2, ?sel_p, !pri_p\} \cup \{(!pri_i, !data_i)\}$. *Some of the pomsets from Figure 4.6 add some order between weakly concurrent actions:* $\eta_1$ *adds causality between weakly concurrent outputs* $!pri_i$ *and* $!data_i$ *from* $\eta_6$*;* $\eta_4$ *also adds direct causalities* $!data_s \leq ?sel_t$ *and* $!data_s \leq ?sel_i$*;* $\eta_5$ *adds causalities* $!data_s \leq ?sel_p$ *and* $!data_s \leq !pri_p$ *from* $\eta_2$*. As strongly concurrent actions remain independent in every case, we can conclude that* $\eta_6 \sqsubseteq \eta_1 \sqsubseteq \eta_4$ *and* $\eta_2 \sqsubseteq \eta_5$*.*

Remark 4. *When there is no weak concurrency, a pomset is only refined by itself, i.e.* $\mathbf{wco} = \emptyset$ *implies* $\eta_1 \sqsubseteq \eta_2$ *iff* $\eta_1 = \eta_2$*.*

Relaxed executions of a system are all the possible refinements of a partial order execution of the system.

**Definition** 4.3 (Relaxed Executions). *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ *with* $C, C' \in \mathcal{C}(\mathcal{E})$ *and* $\eta \in \mathcal{PS}(L)$*, we define*

$$\begin{array}{lll} C \xrightarrow{\eta}_r C' & \triangleq & \exists \eta' \sqsubseteq \eta : C \xrightarrow{\eta'} C' \\ C \xrightarrow{\eta}_r & \triangleq & \exists C' : C \xrightarrow{\eta}_r C' \end{array}$$

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

Example 4.6 (Relaxed Executions). *Behavior* $\eta_6$ *respects the structure of* $\mathcal{S}_3$ *and then we have* $\mathcal{S}_3 \xrightarrow{\eta_6}$*. As the structure of* $\eta_2$ *also respects the structure of* $\mathcal{S}_3$*, we also have* $\mathcal{S}_3 \xrightarrow{\eta_2}$*. Example 4.5 shows that* $\eta_6 \sqsubseteq \eta_1, \eta_6 \sqsubseteq \eta_4$ *and* $\eta_2 \sqsubseteq \eta_5$*, thus we can conclude that*

$$\begin{array}{lll} \mathcal{S}_3 & \xrightarrow{\eta_1}_r & \{\bot, e_1, e_2, e_3, e_4, e_5, e_6, e_9, e_{10}\} \\ \mathcal{S}_3 & \xrightarrow{\eta_4}_r & \{\bot, e_1, e_2, e_3, e_4, e_5, e_6, e_9, e_{10}\} \\ \mathcal{S}_3 & \xrightarrow{\eta_5}_r & \{\bot, e_1, e_2, e_3, e_7, e_8\} \end{array}$$

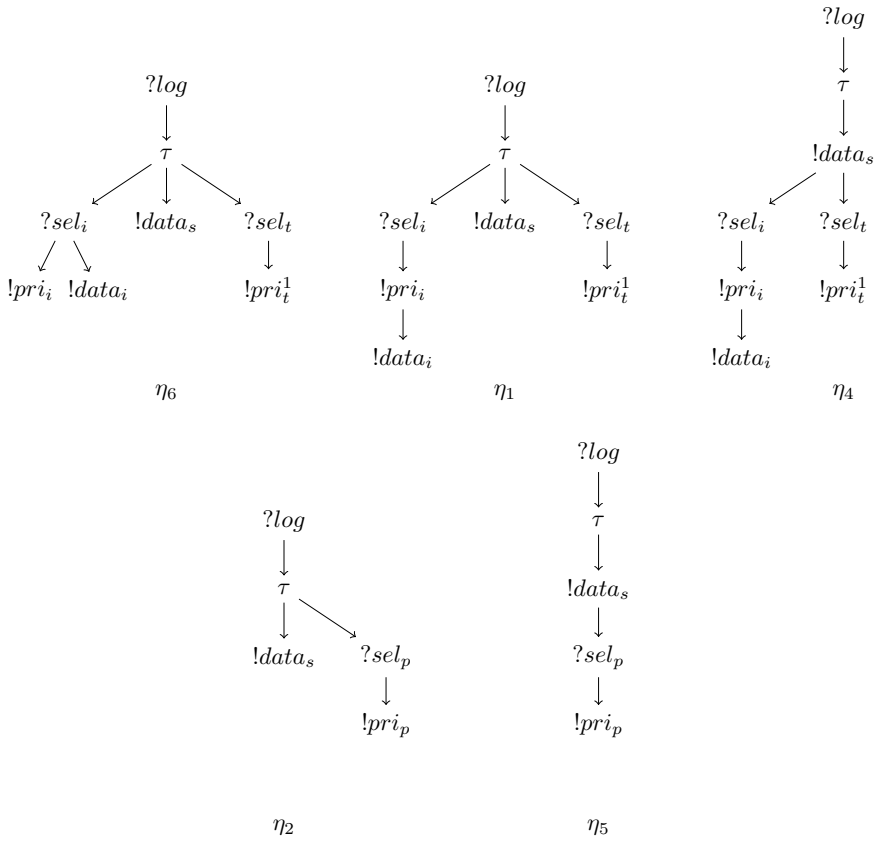



Figure 4.6: Pomset refinement.

*Partial order executions $\eta_1$ and $\eta_4$ show that even if we can interpret the behavior of the system in different ways (due to ordering of weakly concurrent events), the resulting configuration is always the same. As $\sqsubseteq$ is reflexive, we also have $\mathcal{S}_3 \xrightarrow{\eta_6}_r$ and $\mathcal{S}_3 \xrightarrow{\eta_2}_r$.*

Whenever there is no weak concurrency, partial order executions and relaxed executions coincide.

**Proposition** 2. *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ with $C \in \mathcal{C}(\mathcal{E})$ and $\eta \in \mathcal{PS}(L)$ with $\mathbf{wco} = \emptyset$, we have $C \xrightarrow{\eta}$ iff $C \xrightarrow{\eta}_r$.*

*Proof.* Immediate using Remark 4. □

## 4.3 Observing CPOGs

We present in this section, the notions of observations for the weak and strong concurrency semantics.

### 4.3.1 Traces

As in the case of interleaving and partial order semantics, an observation of the system is the $\tau$-abstraction of one of its relaxed executions.

**Definition** 4.4 (Observations). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ with $C, C' \in \mathcal{C}(\mathcal{E})$ and $\omega \in \mathcal{PS}(L)$, we define*

$$\begin{array}{lcl} C \overset{\omega}{\Longrightarrow}_r C' & \triangleq & \exists \eta : C \xrightarrow{\eta}_r C' \text{ and } abs(\eta) = \omega \\ C \overset{\omega}{\Longrightarrow}_r & \triangleq & \exists C' : C \overset{\omega}{\Longrightarrow}_r C' \end{array}$$

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

The definition of traces and reached configurations in this semantics can be easily obtained by replacing $\Longrightarrow$ by $\Longrightarrow_r$ in Definition 3.13.

**Definition** 4.5 (Traces and Reached Configurations). *Let $\mathcal{E} \in \mathcal{ES}(L)$, $\omega \in \mathcal{PS}(L)$ and $C, C' \in \mathcal{C}(\mathcal{E})$, we define*

$$\begin{array}{rcl} traces_r(\mathcal{E}) & \triangleq & \{\omega \in \mathcal{PS}(L) \mid \bot_{\mathcal{E}} \overset{\omega}{\Longrightarrow}_r\} \\ C \ \textbf{\textit{after}}_r \ \omega & \triangleq & \{C' \mid C \overset{\omega}{\Longrightarrow}_r C'\} \end{array}$$

Example 4.7 (Traces and Reached Configurations). *Even if the structure of the system is unique, different observations lead to the same configuration. This is the case for the $\tau$-abstractions of pomsets $\eta_6, \eta_1$ and $\eta_4$ which all lead to the same configuration, i.e. $\mathcal{S}_3$ **after** $abs(\eta_6) = \mathcal{S}_3$ **after** $abs(\eta_1) = \mathcal{S}_3$ **after** $abs(\eta_4) = \{\bot, e_1, e_2, e_3, e_4, e_5, e_6, e_9, e_{10}\}$.*

### 4.3.2 Quiescence and Produced Outputs

The conformance relations presented in Chapter 2 and Chapter 3 compare the produced outputs of the implementation with those of the specification. Conformance of output pomsets is captured by isomorphism under partial order semantics. However, this is not the case in the presence of weak and strong concurrency as we allow the implementation to order some outputs. Produced outputs cannot be directly compared by set inclusion as they are in the case of the **ioco** and **co-ioco** relations. Suppose one has two weakly concurrent outputs $!o_1$ and $!o_2$ depending on input $?i$. After $?i$, the system produces outputs $!o_1$ and $!o_2$ which can be observed concurrently or in any order (due to the relaxed executions). Any produced output in the implementation should refine some produced output in the specification. As both $!o_1 \cdot !o_2$ and $!o_2 \cdot !o_1$ can be inferred from $!o_1$ **wco** $!o_2$ by means of pomset refinement (which is sufficient to compare outputs), we only consider $!o_1$ **wco** $!o_2$ as a produced output, i.e. it is only necessary to consider produced outputs under Definition 3.15.

**Definition** 4.6 (Produced Outputs). *Let $\mathcal{E} \in \mathcal{ES}(L), C \in \mathcal{C}(\mathcal{E})$ and $S \subseteq \mathcal{C}(\mathcal{E})$, we define*

$$\begin{array}{lcl} out_r(C) & \triangleq & out(C) \\ out_r(S) & \triangleq & \bigcup\limits_{C \in S} out_r(C) \end{array}$$

Outputs cannot be compared directly by set inclusion. It is expected that every output produced by the implementation refines some output of the specification. This is captured by the notion of output refinement.

**Definition** 4.7 (Output Refinement). *Let $\mathcal{E} \in \mathcal{ES}(L)$ and $C, C' \in \mathcal{C}(\mathcal{E})$ we define*

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

Since $\sqsubseteq$ is reflexive, an output can be implemented as it is specified. Notice also that the pomset $\delta$ is only refined by itself, therefore if $\delta \in \mathrm{out}_r(C)$ and $\mathrm{out}_r(C) \gg^{-} \mathrm{out}_r(C')$ then $\delta \in \mathrm{out}_r(C')$.

Example 4.8 (Output Refinement). *Consider specification $\mathcal{S}_3$ of Figure 4.5 with configuration $C = \{\bot, e_1, e_3, e_4\}$ and the implementation $\mathcal{I}_7$ of Figure 4.7 with configuration $C' = \{\bot, e_1, e_3, e_4\}$. Both configurations enable the output actions $!pri_i$ and $!data_i$, however we have $out_r(C) = \{!pri_i$ **wco** $!data_i\}$ and $out_r(C') = \{!pri_i \cdot !data_i\}$. Every output of $C$ is refined by an output of $C'$ and thus $out_r(C) \gg^{-} out_r(C')$. If we consider now $\mathcal{S}_3$ with $C = \{\bot, e_1\}$ and $\mathcal{I}_8$ of Figure 4.7 with $C' = \{\bot, e_1\}$ we have that $out(C) = \{!data_s\}$, while $out(C') = \{\delta\}$. The $\delta$ output produced by $C'$ does not refine any output of $C$ and then $out_r(C) \not\gg^{-} out_r(C')$.*

Weak concurrency may introduce extra quiescence in the system. Consider an input $?i$ and an output $!o$ which are specified as weakly concurrent from the

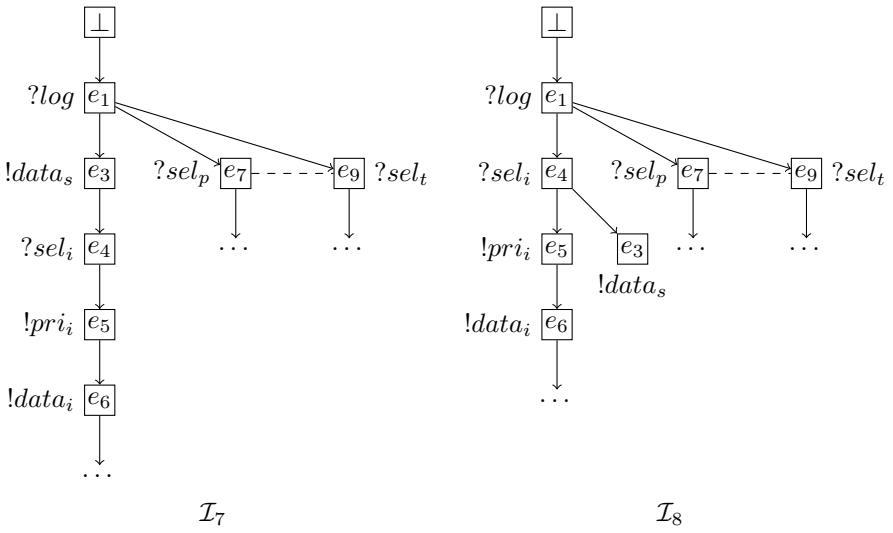



Figure 4.7: Output refinement.

initial configuration. An implementation which orders them as $?i \leq !o$ is quiescent in its initial configuration as only $?i$ is possible. Even if $?i{\cdot}!o$ is a possible relaxed execution in the specification, so is $!o{\cdot}?i$ and therefore some output can be produced from the initial configuration preventing it from being quiescent. We will see that our conformance relation forces any correct implementation that order such kind of actions to produce the output before accepting the input.

When there is no weak concurrency, output refinement boils down to set inclusion.

**Proposition** 3. *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ *with* $\mathbf{wco} = \emptyset$ *and* $C, C' \in \mathcal{C}(\mathcal{E})$, *we have* $out_r(C) \gg^{-} out_r(C') \Leftrightarrow out(C) \subseteq out(C')$.

*Proof.* Immediate using Remark 4. □

### 4.3.3 Possible Inputs

As explained above, some order can be added in an implementation between an input and an output if they are specified as weakly concurrent.

Example 4.9 (Possible Inputs). *Consider the travel agency specification* $\mathcal{S}_3$ *of Figure 4.5 and pomsets* $\eta_6$ *and* $\eta_4$ *in Figure 4.6. In* $\eta_6$, *the selection of insurance is possible after logging in. However, we have seen that* $\eta_4$ *is also a relaxed execution of* $\mathcal{S}_3$ *and in here it is not possible for the user, after just logging in, to select the insurance since the server has not produce the data.*

Consider a pair of weakly concurrent input and output in the specification. If the output precedes the input in the implementation, as this input is considered

as possible in the specification, it needs to be consider also as possible in the implementation even if the output has not been produced yet. This is similar to what happens in remote testing [JJTV99] where communication between test cases and the implementation is asynchronous and a new input can be sent even if an output that precedes it has not been produced yet.

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, it is enough to restrict to possible inputs under partial order semantics as any other possible input can be obtained by pomset refinement.

**Definition** 4.8 (Possible Inputs with Weak Concurrency). *Let $\mathcal{E} \in \mathcal{ES}(L), C \in \mathcal{C}(\mathcal{E})$ and $S \subseteq \mathcal{C}(\mathcal{E})$, we define*

$$\begin{array}{rcl} poss_r(C) & \triangleq & poss(C) \cup poss(C \ \textbf{\textit{after}}\ !\omega) \text{ with } !\omega \in \mathcal{PS}(\mathcal{O}ut) \text{ and } C \stackrel{!\omega}{\Longrightarrow} \\ poss_r(\emptyset) & \triangleq & \mathcal{PS}(\mathcal{I}n) \\ poss_r(S) & \triangleq & \bigcup_{C \in S} poss_r(C) \end{array}$$

Example 4.10 (Possible Inputs with Weak Concurrency). *Since some order can be added between weakly concurrent actions, some inputs are consider as possible even before being actually enabled (it can be seen as if there is some delay before the sending of the input and the actual time the system receives it). Consider Definition 4.8 for possible inputs. After the user logs in and selects an insurance once, it is always possible to select it again even before the outputs are produced, i.e. if $C = (\mathcal{S}_3 \ \textbf{\textit{after}}\ ?log{\cdot}?sel_i)$, then we have $?sel_i \in poss_r(C)$ since $?sel_i \in poss(C \ \textbf{\textit{after}}\ (!pri_i \ \textbf{wco}\ !data_i))$.*

We expect any possible input in the specification to be implemented either as specified or as one of its refinements. However, this is not enough. Suppose there exist two inputs $?i_1$ **wco** $?i_2$ such that $\text{poss}(\mathcal{S}) = \{?i_1, ?i_2, ?i_1 \ \textbf{wco}\ ?i_2\}$ and an implementation that orders them, e.g. $\text{poss}(\mathcal{I}) = \{?i_1, ?i_1{\cdot}?i_2\}$. There is not a possible input of the implementation that refines $?i_2$, for this reason, we restrict to concurrent complete sets of inputs.

**Definition** 4.9 (Concurrent Complete Sets). *Let $\omega \in \mathcal{PS}(L)$ and $C \in \mathcal{C}(\mathcal{E})$, we say that $\omega$ 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 $\omega$*

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

Example 4.11 (Concurrent Complete Sets). *Consider the travel agency $\mathcal{S}_3$ of Figure 4.5 and $C = (\mathcal{S}_3 \ \textbf{\textit{after}}\ ?log{\cdot}!data_s)$. The $?sel_i$ input is possible after logging in and sending the data, i.e. $C \stackrel{?sel_i}{\Longrightarrow}$. However this input is not a concurrent complete set as it can be extended by a concurrent event, i.e. $C \stackrel{\omega}{\Longrightarrow}$ with $\omega = ?sel_i \ \textbf{co}\ ?sel_t$.*

As explained above, the possible inputs of the specification cannot directly be compared with those of the implementation. We want any concurrent complete

input of the specification to be implemented by one of its refinements. This is captured by the notion of input refinement.

**Definition** 4.10 (Input Refinement with Weak Concurrency). *Let* $\mathcal{E} \in \mathcal{ES}(L)$ *with* **wco** $\neq \emptyset$ *and* $C, C' \in \mathcal{C}(\mathcal{E})$*, we define*

$$poss_r(C) \gg^+ poss_r(C') \quad \Leftrightarrow \quad \forall ?\omega \in poss_r(C) : cc(?\omega, C) \text{ implies } \exists ?\omega' \in poss_r(C') : ?\omega \sqsubseteq ?\omega'$$

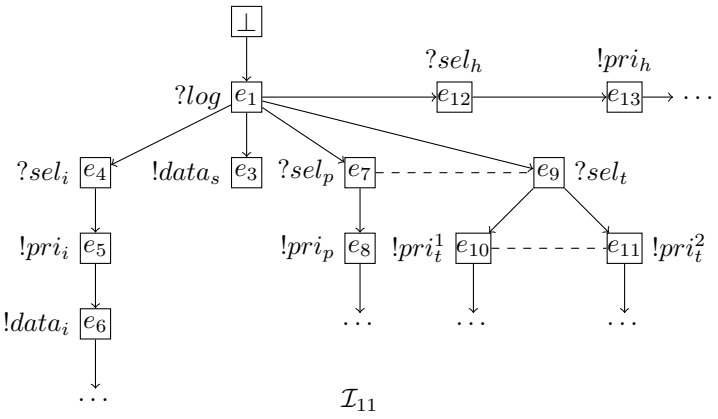



Figure 4.8: Input refinement

Example 4.12 (Input Refinement). *Consider* $\mathcal{S}_3$ *from Figure 4.5 with configuration* $C = \{\bot, e_1\}$ *and implementation* $\mathcal{I}_{11}$ *from Figure 4.8 with* $C' = \{\bot, e_1\}$*. The concurrent complete inputs of* $\mathcal{S}_3$ *in* $C$ *are* $?sel_i$ **co** $?sel_p$ *and* $?sel_i$ **co** $?sel_t$*. As both inputs are implemented, respecting concurrency, in* $\mathcal{I}_{11}$*, we have* $poss_r(C) \gg^+ poss_r(C')$*. Notice that those inputs are not concurrent complete in* $C'$ *(*$?sel_h$ *is still enabled), however, Definition 4.10 forces the completeness of the inputs only in the first configuration.*

We have seen that output refinement boils down to set inclusion where there is no weak concurrency. However this is not true in the case of possible inputs as we consider not only inputs that are enabled in the current configuration, but also those that become enabled after producing some outputs. Those inputs are not supposed to be considered as possible in the case where there is no weak concurrency. For this reason, we propose another definition for input refinement in the case of absence of weak concurrency.

**Definition** 4.11 (Input Refinement without Weak Concurrency). *Let* $\mathcal{E} \in \mathcal{ES}(L)$ *with* **wco** $= \emptyset$ *and* $C, C' \in \mathcal{C}(\mathcal{E})$*, we define*

$$poss_r(C) \gg^+ poss_r(C') \Leftrightarrow \forall ?\omega \in poss(C) : cc(?\omega, C) \text{ implies } ?\omega \in poss(C')$$

## 4.4 The wsc-ioco Conformance Relation

This section presents the testing hypotheses and conformance relation for conformance testing of conditional partial order graphs.

**Testing Hypotheses:** We assume that the specification of the system is given as conditional partial order graph $H = (V, A, X, \phi, \rho)$ over alphabet $L = \mathcal{I}n \uplus \mathcal{O}ut$ of input and output labels. To be able to test an implementation against such a specification, we make the usual testing assumption that the behavior of the SUT itself can be modeled by a conditional partial order graph over the same alphabet of labels. Since projections of a well-formed CPOG are acyclic, it is not necessary to assume the absence of silent or output cycles.

We present now the **wsc-ioco** (weak and strong concurrency) conformance relation that compares produced outputs and possible inputs based on output and input refinements. Refusals and outputs of the implementation must be specified up to refinement. In the case where some order is added by a correct implementation between weakly concurrent input and output, the output must precede the input; this is what we expected as we do not want outputs to depend on extra inputs. Since some inputs are considered as possible even before the system is actually enabled to accept them (see Definition 4.8), inputs do not have extra dependencies either.

This conformance relation is more permissive than **co-ioco** in the sense that some concurrent actions are allowed to be implemented by interleavings. However this is what we expect of actions that were specified as weakly concurrent. This allows the implementation process to be more flexible: if the order between some actions is not decided during the specification of the system, but some order is added during its implementation, this implementation is still conformant w.r.t the requirements.

**Definition** 4.12 (**wsc-ioco**). *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and an implementation of the system, then*

$$\begin{array}{lcll} \mathcal{I} \textbf{ wsc-ioco } \mathcal{S} & \Leftrightarrow & \forall \omega \in traces_r(\mathcal{S}) : & \\ & & poss_r(\mathcal{S} \textbf{ after}_r \ \omega) & \gg^+ \quad poss_r(\mathcal{I} \textbf{ after}_r \ \omega) \\ & & out_r(\mathcal{I} \textbf{ after}_r \ \omega) & \gg^- \quad out_r(\mathcal{S} \textbf{ after}_r \ \omega) \end{array}$$

Example 4.13 (Conformance with Weak Concurrency). *Consider the specification $\mathcal{S}_3$ and implementations $\mathcal{I}_7, \mathcal{I}_8$ from Figure 4.7. Implementation $\mathcal{I}_7$ orders some weakly concurrent events: outputs $!pri_i$ and $!data_i$ are implemented sequentially instead of concurrently, but the output produced ($!pri_i$.$!data_i$) refines an output produced by the specification ($!pri_i$ **wco** $!data_i$). Some order is also added between output $!data_s$ and input $?sel_i$, however, there is not extra causality for the output and the produced outputs in the implementation are those specified. Even if some dependence is added for $?sel_i$, Definition 4.8 considers as possible, inputs that will be enabled after the production of some outputs. We can conclude that $\mathcal{I}_7$* **wsc-ioco** *$\mathcal{S}_3$. Actions $!data_s$ and $?sel_i$*

*are ordered in the opposite way in* $\mathcal{I}_8$. *We have seen in Example 4.8 that* $out_r(\mathcal{I}_8 \ \textbf{\textit{after}}_r \ ?log) \not\gg^{-} out_r(\mathcal{S}_3 \ \textbf{\textit{after}}_r \ ?log)$ *and then* $\neg(\mathcal{I}_8 \ \textbf{wsc-ioco} \ \mathcal{S}_3)$.

**Relating wsc-ioco and co-ioco:** Strong concurrency is forced to be implemented as independence between actions as it is the case for the whole concurrency relation in partial order semantics, therefore, whenever there is no weak concurrency, **wsc-ioco** and **co-ioco** coincide.

**Theorem** 5. *Let* $\mathcal{S}$ *and* $\mathcal{I}$ *be respectively the specification and implementation of a system with* $\textbf{wco} = \emptyset$ *in the specification, then we have*

$$\mathcal{I} \ \textbf{wsc-ioco} \ \mathcal{S} \ \textit{iff} \ \mathcal{I} \ \textbf{co-ioco} \ \mathcal{S}$$

*Proof.* Using Definition 4.11 and Proposition 2 and Proposition 3. □

## 4.5 Conclusion

We have developed a new semantics for concurrent systems that allows to represent both independence between actions and underspecification or refinement; these situations are captured by the notions of strong and weak concurrency. The new semantics generalized both interleaving and partial order semantics: when there is only weak concurrency, the semantics is equivalent to interleavings; while if there is only strong concurrency they are equivalent to partial order semantics.

We propose to use conditional partial order graphs to model the system since they allow to model different scenarios representing both strong and weak concurrency. We gave an unfolding algorithm to convert any conditional partial order graph into an event structure that captures its semantics. This algorithm is based on a SAT encoding of the possible extensions of the system and we showed it is deterministic: the resulting object does not depend on the order in which events are added into the unfolding.

Based on the new semantics, we define new notions of executions and observations and a new conformance relation which boils down to **co-ioco** when there is no weak concurrency (see Theorem 5), but it is more permissive in general: actions specified as (weakly) concurrent can be implemented in a particular order in a correct implementation.

# 5 A Centralized Testing Framework

Ulrich and König [UK99] suggest two test architectures for testing distributed, concurrent systems: a global tester that has total control over the distributed system under test (see Figure 5.1.a) and, more interestingly, a distributed tester comprising several concurrent testers that stimulate the implementation by sending messages on points of control and observation (PCOs) and partially observe the reactions of the implementation on these same PCOs as shown in Figure 5.1.b.

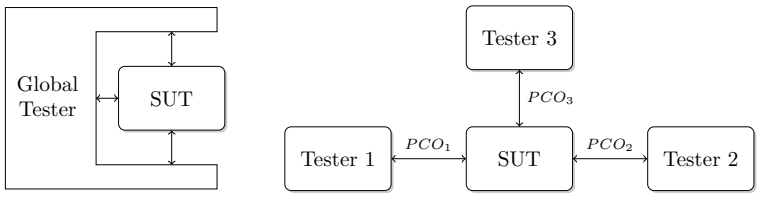



Figure 5.1: The global and distributed testing architectures.

This chapter assumes that global observation and control of the system is possible and proposes a method to generate global test cases. Next chapter deals with a distributed testing architecture.

**Testing Hypotheses:** The previous chapters present three specification languages (LTSs, PNs and CPOGs) together with their corresponding notions of executions, observations and conformance relations. Each of the conformance relations assumes some restrictions on the kind of models that can be used either for the specification or the implementation. The **ioco** relation assumes that the implementation is input-enabled which is not the case for **co-ioco**, however the

later does not accept cycles of outputs in the specification. In order to unify all the frameworks and propose a test generation algorithm that works for all the models, we make the same assumptions over all the models.

We do not restrict to input-enabled implementations (we will use $\mathbf{ioco}_2$ as the conformance relation for LTSs), but we do not accept cycles of outputs and internal actions in the specification. Under these assumptions, we have proved (see Theorem 3 on page 44) that any **co-ioco** conformant implementation passes the **ioco** tests. In addition, **wsc-ioco** and **co-ioco** coincide when weak concurrency is forbidden (see Theorem 5 on page 60). For these reasons, it is sufficient to test the implementations w.r.t the **wsc-ioco** conformance relation and construct test cases for it. The methods we propose to generate test cases do not depend on the different notions of concurrency (concurrency is interpreted under partial order semantics), the difference lies in how those test cases interact with the implementation.

Strongly concurrent actions in a test case should be run concurrently in the implementation, however, if actions were specified as weakly concurrent, the test run is allowed to order them. These differences are captured by the notion of verdicts.

In addition to the notions of global test cases and their interaction with the implementations, this chapter gives sufficient conditions for detecting all and only incorrect implementations and proposes a method to generate test cases.

## 5.1 Global Test Cases, Execution and Verdicts

We define global test cases which can contain concurrency. In practice, such global test cases are not meant to be actually executed globally. They would rather be projected onto the different processes of the distributed system to be executed locally in order to make the observation of concurrency possible (see Chapter 6). Our approach here is to study the testing problem from a centralized point of view, as a basis to the distributed testing problem: the global conformance relations we defined are the relations we want to be able to test in a distributed way (with local control and observation), and the global test cases are the basis for the construction of distributed tests.

### 5.1.1 Global Test Cases

A global 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 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. Similar to this, test cases do not have immediate conflict between outputs and inputs, if not, the implementation may produce the output without allowing the test case to

propose the input. This property is called controllability [JM99]. However, as conflict is inherited w.r.t causality and we accept immediate conflict between outputs, non immediate conflict between inputs or input/output is accepted. Avoiding those immediate conflicts 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 for example, 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 thus require determinism as defined in Definition 3.10 on page 35. 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 where inputs cannot be in immediate conflict.

**Definition** 5.1 (Global Test Cases). *A global test case is an event structure* $\mathcal{T} = (E, \leq, \#, \lambda)$ *such that*

1. $\mathcal{T}$ *is deterministic,*
2. $(E^{\mathcal{I}n} \times E) \cap \#^r = \emptyset$*,*
3. $E$ *is finite.*

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

Example 5.1 (Global Test Cases). *Figure 5.2 presents four event structures.* $\mathcal{T}_1$ *is nondeterministic, i.e. from* $\{\bot, e_1, e_7\}$ *it is possible to perform* $!pri_p$ *and reach both* $\{\bot, e_1, e_7, e_8\}$ *or* $\{\bot, e_1, e_7, e_8^1\}$*;* $\mathcal{T}_2$ *has immediate conflict between actions* $?sel_t$ *and* $?sel_p$*;* $\mathcal{T}_3$ *is infinite. Thus none of* $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ *is a test case. However* $\mathcal{T}_4$ *is finite, deterministic and without inputs in immediate conflict, i.e. it is a test case.*

Allowing to have explicit concurrency in the test cases not only reduces their size by avoiding the state space problem, but it also reduces the number of test cases needed to cover the specification. As concurrency between inputs is interpreted as a nondeterministic choice between the possible interleavings, this choice need to be solved in the test case to avoid uncontrollability.

Example 5.2 (Concurrency in Test Cases). *Consider* $(i_1; o_1 \parallel i_2; o_2)$ *as the specification of the system. Figure 5.3 presents its corresponding global test case (the specification and the test case coincide) and the test cases generated using standard interleaving semantics, i.e. the test suite obtained using the* **ioco** *algorithms (see Section 5.2.1). Not only the size of the global test case is smaller than the size of the ones using interleaving semantics (due to the concurrency between outputs) as it is the case of* $\mathcal{T}_{es}$ *and* $\mathcal{T}^2_{\mathcal{LTS}}, \mathcal{T}^3_{\mathcal{LTS}}$ *where we have four transitions instead of six; but also the size of the test suite generated, i.e. we only have a global test case* $\mathcal{T}_{es}$ *against four test cases in* $\{\mathcal{T}^1_{\mathcal{LTS}}, \mathcal{T}^2_{\mathcal{LTS}}, \mathcal{T}^3_{\mathcal{LTS}}, \mathcal{T}^4_{\mathcal{LTS}}\}$*. The latter is due to the fact that concurrency between inputs is interpreted as a choice in interleaving semantics and this choice need to be solved to overcome controllability problems. Both the size and the number of test cases can be exponentially smaller when concurrency is represented explicitly.*

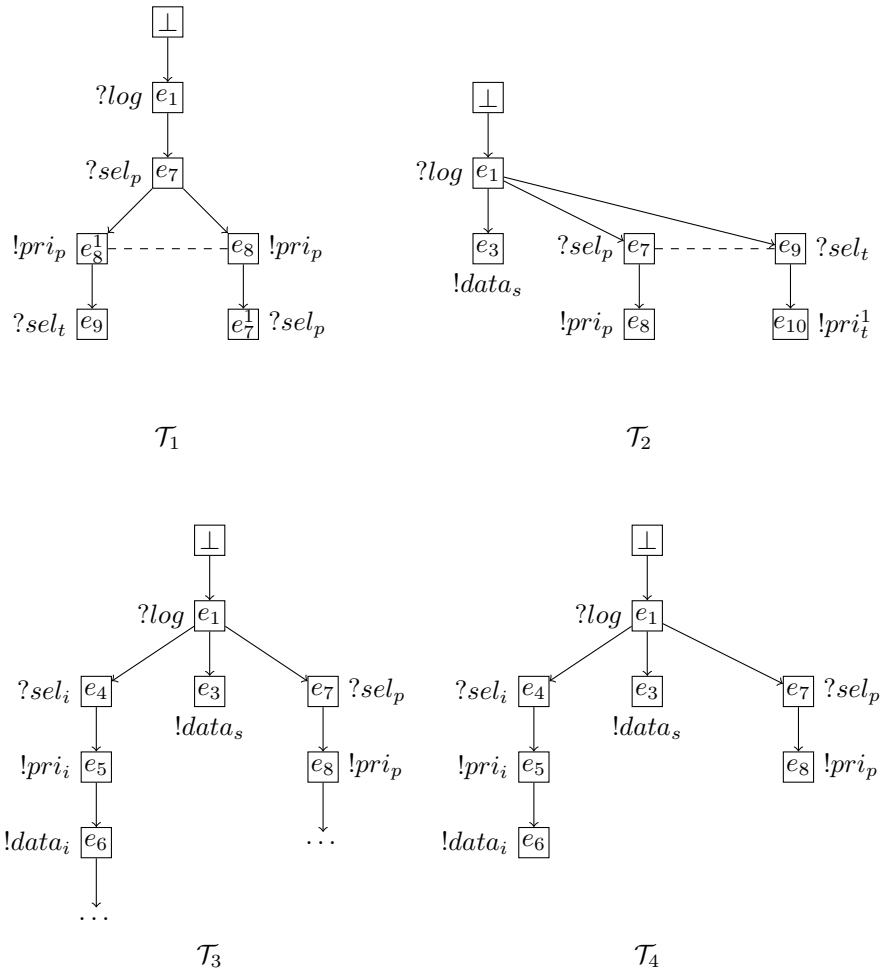



Figure 5.2: Event structures as global test cases.

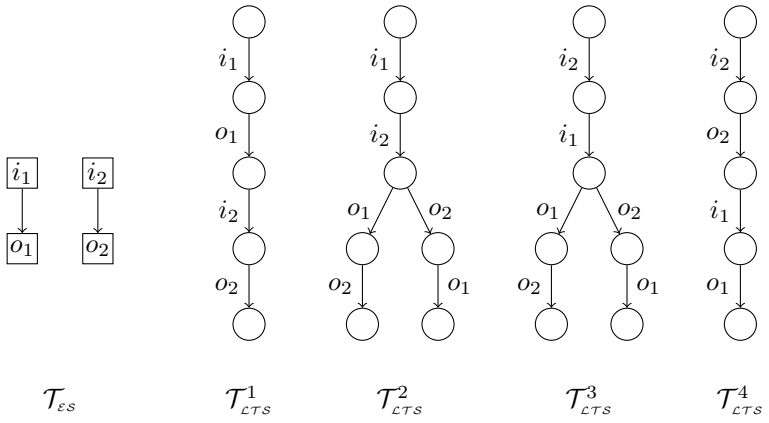



Figure 5.3: Concurrency in test cases.

### 5.1.2 Test Execution

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

In the **ioco** framework, verdicts are modeled by special states of the test case labeled by **pass** and **fail**. A pass verdict can only be reached after observing some output of the implementation (in this framework, $\delta$ is considered an output). When dealing with concurrent systems, one possibility would be to represent verdicts by configurations (via a labeling function for example), but as there is no event labeled by $\delta$, observing $\delta$ does not lead to a new configuration. We model verdicts differently.

Communication between the tester and the implementation can be either synchronous [JJ05, Tre08] or asynchronous [JJKV98]. Parallel composition of labeled transition systems is issued to model test execution when the communication is synchronous and deadlocks of such composition are used to give verdicts about the test run. Such deadlocks are produced in the following situations: *(i)* the implementation proposes an output or $\delta$ action that the test case cannot accept; *(ii)* the test case proposes an input that the implementation cannot accept; or *(iii)* the test case has nothing else to propose (it deadlocks). The first two situations lead to a fail verdict and the latter to a pass one as the experiment is finished.

Test execution assumes that both systems are always prepared to accept an action 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 or sequentially with other outputs). In addition, the product of Petri nets [Win85] does not preserve concurrency and therefore it cannot be used to model test execution. Instead of defining any notion of composition, we propose another approach based on the notion of blocking in the test execution: both systems are executed until their traces differ from each other either by a different input, a different output or the same set of labels forming a different partial order.

After a trace, the test execution can block because of an output or $\delta$ action the implementation produces. This happens if after such an observation the test case cannot accept that action or if the reached configuration is not quiescent respectively, i.e. the implementation produces an output that the test case is not prepared to accept. As we allow weak concurrency, outputs cannot be compared by set inclusion, we need to compare them by output refinement.

**Definition** 5.2 (Output Blocking). *Let $\mathcal{I}, \mathcal{T} \in \mathcal{ES}(L)$ be an implementation and a test case respectively and $\omega \in \mathcal{PS}(L)$, we have*

$$\textbf{\textit{blocks}}_{\mathcal{O}ut}(\mathcal{I}, \mathcal{T}, \omega) \Leftrightarrow out_r(\mathcal{I} \textbf{\textit{ after}}_r \omega) \not\gg^{-} out_r(\mathcal{T} \textbf{\textit{ after}}_r \omega)$$

Example 5.3 (Output Blocking with no Concurrency). *Consider implementations $\mathcal{I}'_2, \mathcal{I}'_3$ and test cases $\mathcal{T}_5, \mathcal{T}_6$ of Figure 5.4. These implementations have the same behavior as $\mathcal{I}_2$ and $\mathcal{I}_3$ of Figure 2.4 on page 24, but they are event structures without concurrency rather than reachability trees. As there is no concurrency, i.e.* $\mathbf{wco} = \emptyset$*, by Proposition 3, output refinement reduces to inclusion of outputs. The trace $?sel_t$ is performed in systems $\mathcal{I}'_2$ and $\mathcal{T}_5$, after it, the execution blocks as the implementation reaches a quiescent configuration, producing a $\delta$ action that is not an output of the test case, i.e. $out_r(\mathcal{I}'_2 \textbf{\textit{ after}}_r ?sel_t) \not\subseteq out_r(\mathcal{T}_5 \textbf{\textit{ after}}_r ?sel_t)$. We have then $\textbf{\textit{blocks}}_{\mathcal{O}ut}(\mathcal{I}'_2, \mathcal{T}_5, ?sel_t)$. The execution between $\mathcal{I}'_3$ and $\mathcal{T}_6$ blocks after $?sel_p$ as the implementation produces action $!pri_p^2$ as an output, but this action is not an output of the test case, thus $out_r(\mathcal{I}'_3 \textbf{\textit{ after}}_r ?sel_p) \not\subseteq out_r(\mathcal{T}_6 \textbf{\textit{ after}}_r ?sel_p)$ and $\textbf{\textit{blocks}}_{\mathcal{O}ut}(\mathcal{I}'_3, \mathcal{T}_6, ?sel_p)$.*

Example 5.4 (Output Blocking with Strong Concurrency Only). *Consider implementations $\mathcal{I}_8, \mathcal{I}_9$ and the test case $\mathcal{T}_7$ from Figure 5.5 where concurrency in the test case is interpreted as strong concurrency, i.e.* $\mathbf{wco} = \emptyset$*. By Proposition 3, output refinement boils down to set inclusion. Test execution between $\mathcal{I}_8$ and $\mathcal{T}_7$ blocks after $?log$ as the implementation reaches a quiescent configuration and produces a $\delta$ action that the test case does not, i.e. $\delta \in out_r(\mathcal{I}_8 \textbf{\textit{ after}}_r ?log)$ while $\delta \notin out_r(\mathcal{T}_7 \textbf{\textit{ after}}_r ?log)$ and $\textbf{\textit{blocks}}_{\mathcal{O}ut}(\mathcal{I}_8, \mathcal{T}_7, ?log)$. After logging in and selecting an insurance, the execution between $\mathcal{I}_9$ and $\mathcal{T}_7$ also blocks as the output produced by the implementation ($!pri_i$* **co** *$!data_i$) is not an output produced by the test case, i.e. $!pri_i$* **co** *$!data_i \notin out_r(\mathcal{T}_7 \textbf{\textit{ after}}_r ?log \cdot ?sel_i) = \{!pri_i \cdot !data_i\}$ and therefore $\textbf{\textit{blocks}}_{\mathcal{O}ut}(\mathcal{I}_9, \mathcal{T}_7, ?log \cdot ?sel_i)$.*

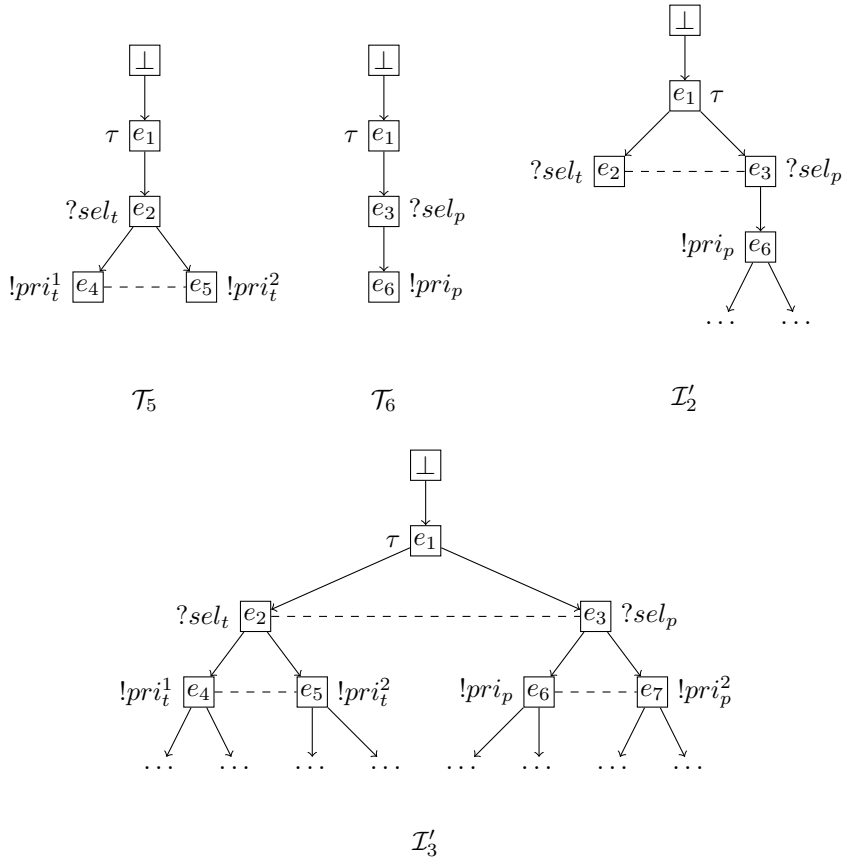



Figure 5.4: Output blocking with no concurrency.

Example 5.5 (Test Execution with Weak Concurrency). *Consider implementation* $\mathcal{I}_7$ *and the test case* $\mathcal{T}_8$ *from Figure 5.6 where concurrency between actions* $?sel_i, !pri_i, !data_i$ *and* $!data_s$ *is considered as weak, i.e.* $\{(!pri_i, !data_i), (!data_s, ?sel_i), (!data_s, !pri_i), (!data_s, !data_i)\} \subseteq \mathbf{wco}$. *The produced outputs after logging in coincide in both event structures, i.e.* $out_r(\mathcal{I}_7 \ \textbf{\textit{after}}_r \ ?log) = \{!data_s\} = out_r(\mathcal{T}_8 \ \textbf{\textit{after}}_r \ ?log)$, *and therefore the test execution does not block. Even if the outputs produced by the implementation and the test case after logging in and selecting an insurance do not coincide, i.e.* $out_r(\mathcal{I}_7 \ \textbf{\textit{after}}_r \ ?log{\cdot}?sel_i) = \{!pri_i{\cdot}!data_i\} \neq \{!pri_i \ \mathbf{wco} \ !data_i\} = out_r(\mathcal{T}_8 \ \textbf{\textit{after}}_r \ ?log{\cdot}?sel_i)$, *the output of the implementation refines the output of the test case and therefore the execution does not block either.*

The other blocking situation occurs when the test case can propose a concurrent complete set of inputs that the implementation is not prepared to accept.

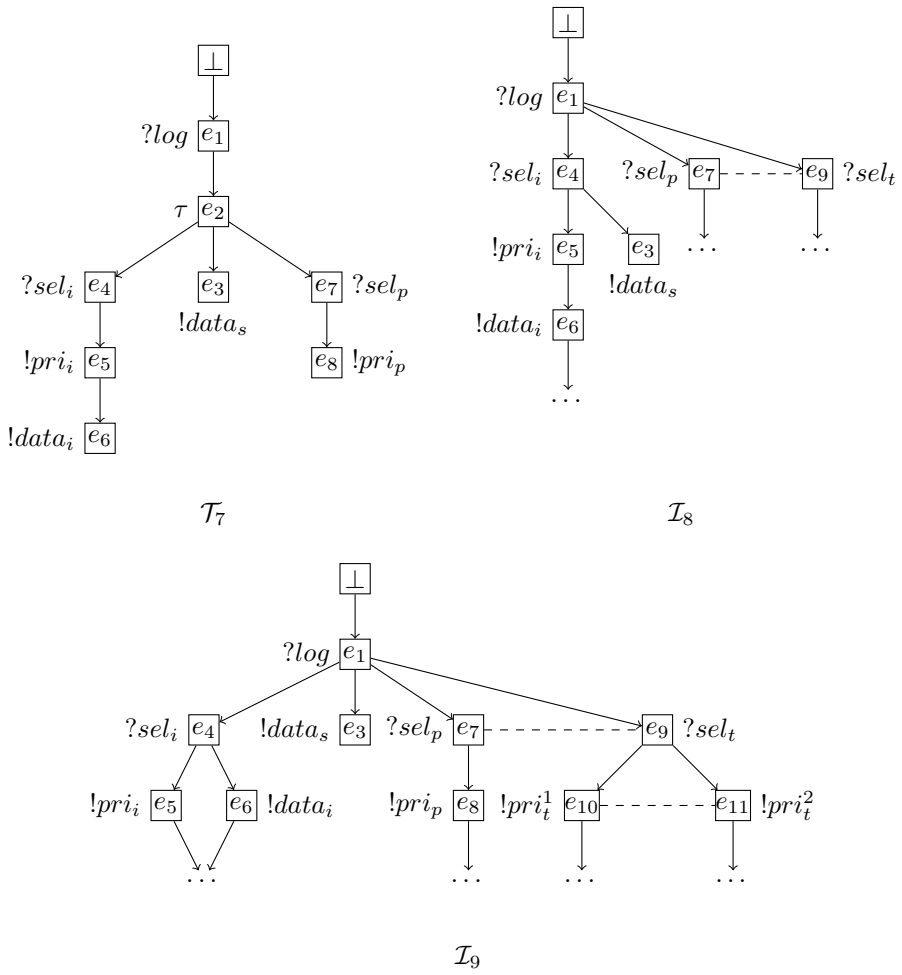



Figure 5.5: Output blocking with strong concurrency only.

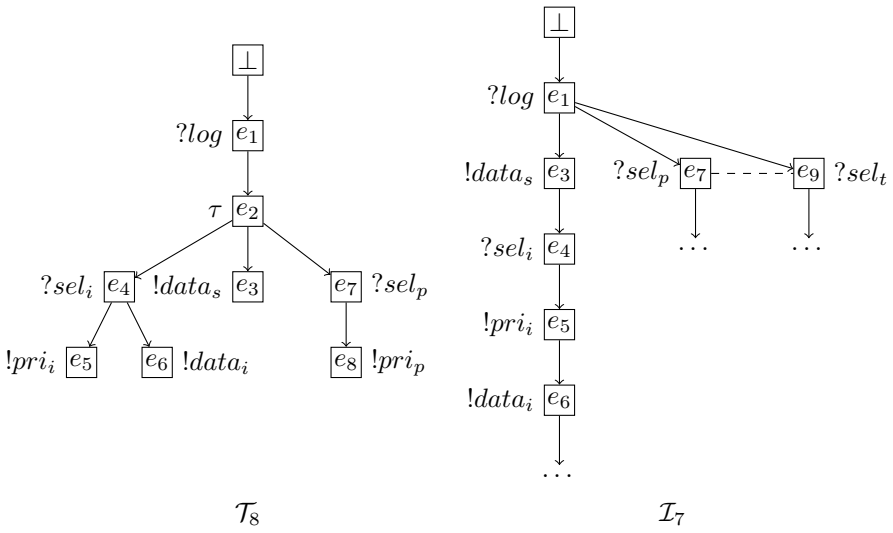



Figure 5.6: Test execution with weak concurrency.

**Definition** 5.3 (Input Blocking). *Let $\mathcal{I}, \mathcal{T} \in \mathcal{ES}(L)$ and $\omega \in \mathcal{PS}(L)$, we have*

$$\textbf{\textit{blocks}}_{\mathcal{I}n}(\mathcal{I}, \mathcal{T}, \omega) \Leftrightarrow \textit{poss}_r(\mathcal{T} \textbf{\textit{ after}}_r\ \omega) \not\gg^+ \textit{poss}_r(\mathcal{I} \textbf{\textit{ after}}_r\ \omega)$$

Example 5.6 (Input Blocking by Missing Input). *Consider the test case $\mathcal{T}_9$ and implementation $\mathcal{I}_6$ from Figure 5.7. After logging in, the test case proposes a concurrent complete input that is not possible in the implementation (neither one of its refinement), i.e. $?sel_i$ **co** $?sel_t \in \textit{poss}_r(\mathcal{T}_9 \textbf{\textit{ after}}_r\ ?log)$. The possible inputs of $\mathcal{I}_6$ are $\textit{poss}_r(\mathcal{I}_6 \textbf{\textit{ after}}_r\ ?log) = \{?sel_i, ?sel_p, ?sel_i \textbf{ co } ?sel_p\}$ and none of those inputs refines $?sel_i$ **co** $?sel_t$. This leads to an input blocking, i.e. $\textbf{\textit{blocks}}_{\mathcal{I}n}(\mathcal{I}_6, \mathcal{T}_9, ?log)$.*

Example 5.7 (Input Blocking with Strong Concurrency Only). *Consider the test case $\mathcal{T}_8$ and implementation $\mathcal{I}_7$ from Figure 5.6 where concurrency of the test case is interpreted as strong concurrency, i.e. $\textbf{wco} = \emptyset$. Using Definition 4.11, input refinement reduces to set inclusion of concurrent complete inputs that are enabled in the current configuration. After logging in, the test case proposes a concurrent complete input that is not possible in the implementation $\mathcal{I}_7$, i.e. $?sel_i$ **co** $?sel_p \in \textit{poss}(\mathcal{T}_8 \textbf{\textit{ after}}_r\ ?log)$, while $\textit{poss}(\mathcal{I}_7 \textbf{\textit{ after}}_r\ ?log) = \{?sel_t, ?sel_p\}$, therefore $\textbf{\textit{blocks}}_{\mathcal{I}n}(\mathcal{I}_7, \mathcal{T}_8, ?log)$.*

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

**Definition** 5.4 (Verdicts). *Let $\mathcal{I}$ be an implementation, and $T$ a test suite, we have:*

$$\mathcal{I} \textbf{ fail } T \Leftrightarrow \exists \mathcal{T} \in T, \omega \in \textit{traces}_r(\mathcal{T}) : \textbf{\textit{blocks}}_{\mathcal{O}ut}(\mathcal{I}, \mathcal{T}, \omega) \vee \textbf{\textit{blocks}}_{\mathcal{I}n}(\mathcal{I}, \mathcal{T}, \omega)$$

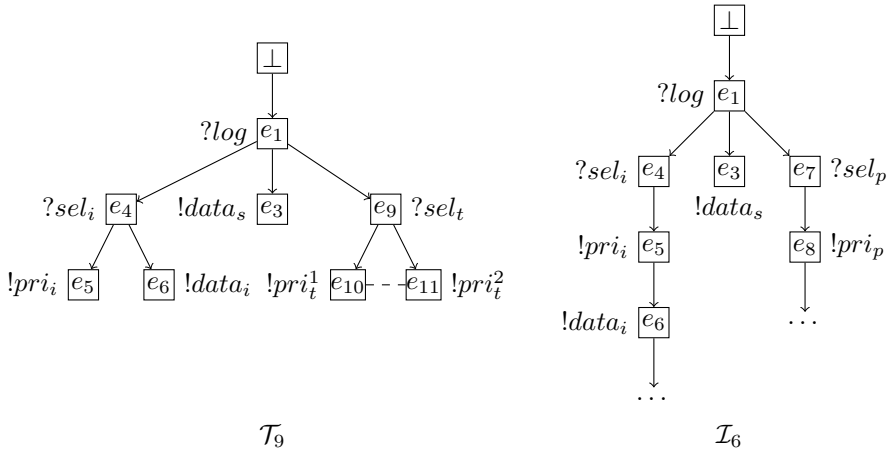



Figure 5.7: Input Blocking by removing inputs.

*If the implementation does not fail the test suite, then it passes it, denoted by* $\mathcal{I}$ **pass** $T$.

Example 5.8 (Verdicts). *Consider the test suite* $T = \{\mathcal{T}_5, \mathcal{T}_6, \mathcal{T}_7, \mathcal{T}_8, \mathcal{T}_9\}$. *In the examples above we saw that the executions of implementations* $\mathcal{I}_2', \mathcal{I}_3', \mathcal{I}_6, \mathcal{I}_8, \mathcal{I}_9$ *and some of the test cases in* $T$ *block. We can conclude that those implementations fail the test suite* $T$. *The verdict of the interaction between* $\mathcal{I}_7$ *and* $\mathcal{T}_8$ *depends on the presence or not of weak concurrency as in one case the execution blocks (see Example 5.7) while in the other not (see Example 5.5). If* $\{(!pri_i, !data_i), (!data_s, ?sel_i), (!data_s, !pri_i), (!data_s, !data_i)\} \subseteq$ **wco**, *then we have* $\mathcal{I}_7$ **pass** $T$, *but whenever* **wco** $= \emptyset$, *we have* $\mathcal{I}_7$ **fail** $T$.

### 5.1.3 Completeness of the Test Suite

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 which accepts only implementations where two events are concurrent and rejects those implementations ordering the events, even if those events were specified as weakly concurrent and those implementations are correct w.r.t **wsc-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** 5.5 (Properties of Test Suites). *Let* $\mathcal{S}$ *be a specification and* $T$ *a test*

*suite, then*

$$\begin{array}{llllll} T \textit{ is sound} & \triangleq & \forall \mathcal{I}: & \mathcal{I} \textbf{ fail } T & \textit{implies} & \neg(\mathcal{I} \textbf{ wsc-ioco } \mathcal{S}) \\ T \textit{ is exhaustive} & \triangleq & \forall \mathcal{I}: & \mathcal{I} \textbf{ fail } T & \textit{if} & \neg(\mathcal{I} \textbf{ wsc-ioco } \mathcal{S}) \\ T \textit{ is complete} & \triangleq & \forall \mathcal{I}: & \mathcal{I} \textbf{ fail } T & \textit{iff} & \neg(\mathcal{I} \textbf{ wsc-ioco } \mathcal{S}) \end{array}$$

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

**Theorem** 6. *Let $\mathcal{S} \in \mathcal{ES}(L)$ be a specification and $T$ a test suite such that*

*a)* $\forall \mathcal{T} \in T : traces_r(\mathcal{T}) \subseteq traces_r(\mathcal{S})$

*b)* $\forall \mathcal{T} \in T, \omega \in traces_r(\mathcal{T}) : out_r(\mathcal{S} \textbf{ after}_r\ \omega) \subseteq out_r(\mathcal{T} \textbf{ after}_r\ \omega)$

*c)* $\forall \mathcal{T} \in T, \omega \in traces_r(\mathcal{T}) : cc(?\omega, \mathcal{T} \textbf{ after}_r\ \omega) \Rightarrow cc(?\omega, \mathcal{S} \textbf{ after}_r\ \omega)$

*then $T$ is sound for $\mathcal{S}$ w.r.t* **wsc-ioco**.

Notice that the trace inclusion required in a) ensures that any possible input in the test case is also possible in the specification (either considering *poss* or $poss_r$).

*Proof.* $T$ is sound for $\mathcal{S}$ w.r.t. **wsc-ioco** iff for every implementation $\mathcal{I}$ that fails the test suite, we have that it does not conform to the specification. We assume $\mathcal{I}$ **fail** $T$ and by Definition 5.4 we have:

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

and at least one of the following cases holds:

*(i)* The test execution blocks after $\omega$ because of an output produced by the implementation,

$$\begin{array}{l} \quad \exists \omega \in \text{traces}_r(\mathcal{T}) : \textbf{blocks}_{\mathcal{O}ut}(\mathcal{I}, \mathcal{T}, \omega) \\ \Rightarrow \{* \text{ Definition 5.2 } *\} \\ \quad \exists \omega \in \text{traces}_r(\mathcal{T}) : \text{out}_r(\mathcal{I} \textbf{ after}_r\ \omega) \not\gg^{-} \text{out}_r(\mathcal{T} \textbf{ after}_r\ \omega) \\ \Rightarrow \{* \text{ Definition 4.7 } *\} \\ \quad \exists \omega \in \text{traces}_r(\mathcal{T}) : \exists x \in \text{out}_r(\mathcal{I} \textbf{ after}_r\ \omega) : \forall x' \in \text{out}_r(\mathcal{T} \textbf{ after}_r\ \omega) : \ x' \not\sqsubseteq\ x \\ \Rightarrow \{* \text{ Assumptions a) and b) } *\} \\ \quad \exists \omega \in \text{traces}_r(\mathcal{S}) : \exists x \in \text{out}_r(\mathcal{I} \textbf{ after}_r\ \omega) : \forall x' \in \text{out}_r(\mathcal{S} \textbf{ after}_r\ \omega) : \ x' \not\sqsubseteq\ x \\ \Rightarrow \{* \text{ Definition 4.7 } *\} \\ \quad \exists \omega \in \text{traces}_r(\mathcal{S}) : \text{out}_r(\mathcal{I} \textbf{ after}_r\ \omega) \not\gg^{-} \text{out}_r(\mathcal{S} \textbf{ after}_r\ \omega) \\ \Rightarrow \{* \text{ Definition 4.12 } *\} \\ \quad \neg(\mathcal{I} \textbf{ wsc-ioco } \mathcal{S}) \end{array}$$

*(ii)* The test execution blocks after $\omega$ because of an input proposed by the test case,

$$\begin{array}{ll} & \exists \omega \in \text{traces}_r(\mathcal{T}) : \textbf{blocks}_{\mathcal{I}n}(\mathcal{I}, \mathcal{T}, \omega) \\ \Rightarrow & \{* \text{ Definition 5.3 } *\} \\ & \exists \omega \in \text{traces}_r(\mathcal{T}) : \text{poss}_r(\mathcal{T} \textbf{ after}_r\ \omega) \not\gg^{+} \text{poss}_r(\mathcal{I} \textbf{ after}_r\ \omega) \end{array}$$

As the definition of input refinement depends on the presence or not of weak concurrency, we split the proof by cases.

if $\mathbf{wco} \neq \emptyset$

$$
\begin{array}{ll}
\Rightarrow & \{* \text{ Definition 4.10 } *\} \\
& \exists \omega \in \text{traces}_r(\mathcal{T}) : \exists ?\omega \in \text{poss}_r(\mathcal{T} \textbf{ after}_r \omega) : \text{cc}(?\omega, \mathcal{T} \textbf{ after}_r \omega) \\
& \wedge \forall ?\omega' \in \text{poss}_r(\mathcal{I} \textbf{ after}_r \omega) : !\omega \not\sqsubseteq !\omega' \\
\Rightarrow & \{* \text{ Assumptions a) and c) } *\} \\
& \exists \omega \in \text{traces}_r(\mathcal{S}) : \exists ?\omega \in \text{poss}_r(\mathcal{S} \textbf{ after}_r \omega) : \text{cc}(?\omega, \mathcal{S} \textbf{ after}_r \omega) \\
& \wedge \forall ?\omega' \in \text{poss}_r(\mathcal{I} \textbf{ after}_r \omega) : !\omega \not\sqsubseteq !\omega' \\
\Rightarrow & \{* \text{ Definition 4.10 } *\} \\
& \exists \omega \in \text{traces}_r(\mathcal{S}) : \text{poss}_r(\mathcal{S} \textbf{ after}_r \omega) \not\gg^{+} \text{poss}_r(\mathcal{I} \textbf{ after}_r \omega) \\
\Rightarrow & \{* \text{ Definition 4.12 } *\} \\
& \neg(\mathcal{I} \textbf{ wsc-ioco } \mathcal{S})
\end{array}
$$

if $\mathbf{wco} = \emptyset$

$$
\begin{array}{ll}
\Rightarrow & \{* \text{ Definition 4.11 } *\} \\
& \exists \omega \in \text{traces}_r(\mathcal{T}) : \exists ?\omega \in \text{poss}(\mathcal{T} \textbf{ after}_r \omega) : \text{cc}(?\omega, \mathcal{T} \textbf{ after}_r \omega) \\
& \wedge ?\omega \notin \text{poss}(\mathcal{I} \textbf{ after}_r \omega) \\
\Rightarrow & \{* \text{ Assumptions a) and c) } *\} \\
& \exists \omega \in \text{traces}_r(\mathcal{S}) : \exists ?\omega \in \text{poss}(\mathcal{S} \textbf{ after}_r \omega) : \text{cc}(?\omega, \mathcal{S} \textbf{ after}_r \omega) \\
& \wedge ?\omega \notin \text{poss}(\mathcal{I} \textbf{ after}_r \omega) : \\
\Rightarrow & \{* \text{ Definition 4.11 } *\} \\
& \exists \omega \in \text{traces}_r(\mathcal{S}) : \text{poss}_r(\mathcal{S} \textbf{ after}_r \omega) \not\gg^{+} \text{poss}_r(\mathcal{I} \textbf{ after}_r \omega) \\
\Rightarrow & \{* \text{ Definition 4.12 } *\} \\
& \neg(\mathcal{I} \textbf{ wsc-ioco } \mathcal{S})
\end{array}
$$

□

Example 5.9 (Soundness of the Test Suite). *Consider the specification of the ticket provider $\mathcal{S}_1 = \mathcal{R}_{tick}$ from Figure 2.1 and test cases $\mathcal{T}_5, \mathcal{T}_6$ from Figure 5.4. It is easy to see that both test cases fulfill the assumptions of Theorem 6 and we can conclude that they form a sound test suite. The execution of both test cases with implementations $\mathcal{I}_2', \mathcal{I}_3'$ in Figure 5.4 blocks and both implementations fail the test suite $\{\mathcal{T}_5, \mathcal{T}_6\}$. Therefore they do not conform to the specification, which is consistent with the result of Example 2.11. If we consider $\mathcal{S}_2 = \mathcal{E}_{ag}$ from Figure 3.3 as the specification of the system, by Theorem 6, the test case $\mathcal{T}_7$ from Figure 5.5 forms a sound test suite. As the execution of this test case with both $\mathcal{I}_8$ and $\mathcal{I}_9$ from Figure 5.5 blocks, both implementations are non conformant (as shown in Example 3.13). Consider Figure 5.7, the test case $\mathcal{T}_9$ also fulfills the assumptions of the theorem when one considers $\mathcal{S}_3$ as the specification of the system, thus $\{\mathcal{T}_9\}$ is a sound test suite. It is shown in Example 5.6 that the execution of this test and implementation $\mathcal{I}_6$ blocks and therefore the implementation does not conform to the specification as in Example 3.12.*

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

**Theorem** 7. *Let $\mathcal{S} \in \mathcal{ES}(L)$ be a specification and $T$ a test suite such that*

*a)* $\forall \omega \in traces_r(\mathcal{S}) : \exists \mathcal{T} \in T : \omega \in traces_r(\mathcal{T})$

*b)* $\forall \mathcal{T} \in T, \omega \in traces_r(\mathcal{T}) : out_r(\mathcal{T}\ \textbf{\textit{after}}_r\ \omega) \subseteq out_r(\mathcal{S}\ \textbf{\textit{after}}_r\ \omega)$

*c)* $\forall \mathcal{T} \in T, \omega \in traces_r(\mathcal{T}) : cc(?\omega, \mathcal{S}\ \textbf{\textit{after}}_r\ \omega) \Rightarrow cc(?\omega, \mathcal{T}\ \textbf{\textit{after}}_r\ \omega)$

*then $T$ is exhaustive for $\mathcal{S}$ w.r.t* **wsc-ioco**.

*Proof.* We need to prove that if $\mathcal{I}$ does not conform to $\mathcal{S}$ then $\mathcal{I}$ **fail** $T$. We assume $\neg(\mathcal{I}$ **wsc-ioco** $\mathcal{S})$, and then at least one of the following two cases holds:

*(i)* The implementation does not conform to the specification because an output produced by the implementation does not refine any specified output:

$\exists \omega \in \text{traces}_r(\mathcal{S}) : \text{out}_r(\mathcal{I}\ \textbf{after}_r\ \omega) \not\gg^{-} \text{out}_r(\mathcal{S}\ \textbf{after}_r\ \omega)$
$\Rightarrow$ {∗ Definition 4.7 ∗}
$\exists \omega \in \text{traces}_r(\mathcal{S}) : \exists x \in \text{out}_r(\mathcal{I}\ \textbf{after}_r\ \omega) : \forall x' \in \text{out}_r(\mathcal{S}\ \textbf{after}_r\ \omega) : x' \not\sqsubseteq\ x$
$\Rightarrow$ {∗ Assumptions a) and b) ∗}
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \exists x \in \text{out}_r(\mathcal{I}\ \textbf{after}_r\ \omega) : \forall x' \in \text{out}_r(\mathcal{T}\ \textbf{after}_r\ \omega) :$
$x' \not\sqsubseteq\ x$
$\Rightarrow$ {∗ Definition 4.7 ∗}
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \text{out}_r(\mathcal{I}\ \textbf{after}_r\ \omega) \not\gg^{-} \text{out}_r(\mathcal{T}\ \textbf{after}_r\ \omega)$
$\Rightarrow$ {∗ Definition 5.2 ∗}
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \textbf{blocks}_{\mathcal{O}ut}(\mathcal{I}, \mathcal{T}, \omega)$
$\Rightarrow$ {∗ Definition 5.4 ∗}
$\mathcal{I}$ **fail** $T$

*(ii)* The implementation does not conform to the specification because an input from the specification is not possible in the implementation:

$$\exists \omega \in \text{traces}_r(\mathcal{S}) : \text{poss}_r(\mathcal{S}\ \textbf{after}_r\ \omega) \not\gg^{+} \text{poss}_r(\mathcal{I}\ \textbf{after}_r\ \omega)$$

We split the proof depending on the presence or not of weak concurrency.

if **wco** $\neq \emptyset$

$\Rightarrow$ {∗ Definition 4.10 ∗}
$\exists \omega \in \text{traces}_r(\mathcal{S}) : \exists ?\omega \in \text{poss}_r(\mathcal{S}\ \textbf{after}_r\ \omega) : \text{cc}(?\omega, \mathcal{S}\ \textbf{after}_r\ \omega)$
$\wedge \forall ?\omega' \in \text{poss}_r(\mathcal{I}\ \textbf{after}_r\ \omega) : !\omega \not\sqsubseteq\ !\omega'$
$\Rightarrow$ {∗ Assumptions a) and c) ∗}
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \exists ?\omega \in \text{poss}_r(\mathcal{T}\ \textbf{after}_r\ \omega) : \text{cc}(?\omega, \mathcal{T}\ \textbf{after}_r\ \omega)$
$\wedge \forall ?\omega' \in \text{poss}_r(\mathcal{I}\ \textbf{after}_r\ \omega) : !\omega \not\sqsubseteq\ !\omega'$
$\Rightarrow$ {∗ Definition 4.10 ∗}
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \text{poss}(\mathcal{T}\ \textbf{after}_r\ \omega) \not\gg^{+} \text{poss}_r(\mathcal{I}\ \textbf{after}_r\ \omega)$
$\Rightarrow$ {∗ Definition 5.3 ∗}
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \textbf{blocks}_{\mathcal{I}n}(\mathcal{I}, \mathcal{T}, \omega)$
$\Rightarrow$ {∗ Definition 5.4 ∗}
$\mathcal{I}$ **fail** $T$

if $\mathbf{wco} = \emptyset$

$\Rightarrow$ $\{*$ Definition 4.11 $*\}$
$\exists \omega \in \text{traces}_r(\mathcal{S}) : \exists ?\omega \in \text{poss}(\mathcal{S}\ \mathbf{after}_r\ \omega) : \text{cc}(?\omega, \mathcal{S}\ \mathbf{after}_r\ \omega)$
$\wedge ?\omega \in \text{poss}(\mathcal{I}\ \mathbf{after}_r\ \omega)$

$\Rightarrow$ $\{*$ Assumptions a) and c) $\}*$
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \exists ?\omega \in \text{poss}(\mathcal{T}\ \mathbf{after}_r\ \omega) : \text{cc}(?\omega, \mathcal{T}\ \mathbf{after}_r\ \omega)$
$\wedge ?\omega \in \text{poss}(\mathcal{I}\ \mathbf{after}_r\ \omega)$

$\Rightarrow$ $\{*$ Definition 4.11 $*\}$
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \text{poss}_r(\mathcal{T}\ \mathbf{after}_r\ \omega) \not\gg^{+} \text{poss}_r(\mathcal{I}\ \mathbf{after}_r\ \omega)$

$\Rightarrow$ $\{*$ Definition 5.3 $*\}$
$\exists \mathcal{T} \in T, \omega \in \text{traces}_r(\mathcal{T}) : \mathbf{blocks}_{\mathcal{I}n}(\mathcal{I}, \mathcal{T}, \omega)$

$\Rightarrow$ $\{*$ Definition 5.4 $*\}$
$\mathcal{I}\ \mathbf{fail}\ T$

□

## 5.2 Constructing Test Cases

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

### 5.2.1 Test Derivation for LTSs

In the **ioco** theory, the behavior of a test case is described by a finite tree with verdicts in the leaves. In the test suite generated by the algorithm proposed by Tretmans [Tre08], in each internal node of a test case, either one specific input action and all possible outputs can occur, or every outputs and the special action $\theta$ can occur. The special label $\theta \notin L \cup \{\delta\}$ is used to detect quiescent states of an implementation, so it can be thought of as the communicating counterpart of a $\delta$ action.

If the test case is in a state where an input and all the outputs are possible and the implementation produces some output, the execution synchronizes, forbidding the tester to propose the input. The authors of [JM99] argue that a test case should not allow this kind of choices; they call this property controllability. The test case generation algorithm they propose takes as an input the specification of the system and a finite automata (called test purpose) representing the behaviors to be tested. The product of those systems is computed, extended by $\delta$ actions and determinized, producing the complete test graph (CTG): a graph which contains all the test cases corresponding to the given test purpose and where some of its states are labeled by verdicts. The complete test graph is still uncontrollable, but it can be pruned by a backtrack strategy from a **pass** state (chosen nondeterministically) to the initial state, producing a test case.

### 5.2.2 Test Derivation for ESs

As a Petri net cannot be determinized [GG99], we assume that the specification of the system is given as a deterministically labeled Petri net which unfolds into a deterministic event structure. In order to obtain a test case, we need to truncate the unfolding into a finite prefix (we explain how to do this in Section 5.3) and deal with uncontrollability.

The authors of [JM99] allow immediate conflict between inputs and outputs and deals with them by a backtrack strategy. If we follow the same strategy, when an input is selected, outputs are discarded. This implies that some test cases do not preserve outputs of the specification and therefore Theorem 6 cannot be applied to prove soundness of the test suite. For this reason, we avoid immediate conflict between inputs and outputs in the specification.

**Assumption** 2. *We will only consider specifications where there is no immediate conflict between input and output events, i.e.* $\forall M \in \mathcal{R}(\mathcal{N}), t_1 \in T^{\mathcal{I}n}, t_2 \in T^{\mathcal{O}ut}$, $M \xrightarrow{t_1} \wedge\ M \xrightarrow{t_2}$ *implies* ${}^\bullet t_1 \cap {}^\bullet t_2 = \emptyset$. *This implies that the unfolding of the net is such that* $\forall e_1 \in E^{\mathcal{I}n}, e_2 \in E^{\mathcal{O}ut} : \neg(e_1 \#^r e_2)$.

Immediate conflict between inputs still need to be solved. One possible way of obtaining a test case is to traverse the prefix and add events to the test case as far as they do not introduce immediate conflict with an input that is already part of the test case. Algorithm 3 builds a global test case from a finite event structure by resolving immediate conflicts between inputs, while accepting several branches in case of conflict between outputs (note that "mixed" immediate conflicts between inputs and outputs have been ruled out by Assumption 2). At the end of the algorithm, all such conflicts have been resolved in one way, following one fixed strategy of resolution of immediate input conflicts. Each strategy can be represented as a linearization of the causality relation that specifies in which order the events are selected by the algorithm. However, as it can be seen in Section 5.2.4, the number of linearizations needed to cover the prefix is bounded by the number of direct conflicts between inputs.

Analogously to the nondeterministic choice of the next input in the algorithm by Tretmans or the nondeterministic choice in the **pass** state to start the backtrack strategy, we assume a linearization is selected non deterministically. The algorithms for LTSs build a test case that accepts any output the implementation may produce returning a pass verdict if the output was specified and a fail one if not. Contrary to this, we neither complete the specification nor label it with verdicts; we build a test case that only accepts those outputs that were specified. Finally, the algorithm by Tretmans allows to chose for termination while finiteness of the test case using a backtrack strategy is given by the finiteness of the path between the chosen **pass** state and the initial state. Termination of our algorithm is given by the finiteness of the event structure that is given as an input to the algorithm.

Example 5.10 (Test Derivation for ESs). *Consider the event structure Fin of Figure 5.8 and linearization* $\mathcal{R}_1 = \bot \cdot e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_7 \cdot e_8 \cdot e_9 \cdot e_{10} \cdot e_{11}$ *are*

**Algorithm 3** Immediate input conflict solver

**Require:** A finite and deterministic $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ such that $\forall e \in E^{\mathcal{I}n}, e' \in E^{\mathcal{O}ut} : \neg(e \#^r e')$ and a linearization $\mathcal{R}$ of $\leq$

**Ensure:** A test case $\mathcal{T}$ such that $\forall \omega \in \text{traces}_r(\mathcal{T}) : \text{out}_r(\mathcal{T} \textbf{ after}_r \omega) = \text{out}_r(\mathcal{E} \textbf{ after}_r \omega)$

$E_{\mathcal{T}} := \emptyset$
$E_{temp} := E$
**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** $(\{e_m\} \times E^{\mathcal{I}n}_{\mathcal{T}}) \cap \#^r = \emptyset \wedge \langle e_m \rangle \subseteq E_{\mathcal{T}}$ **then**
    /* the current event $e_m$ is not in conflict with any event of the prefix and its past is already in the prefix */
    $E_{\mathcal{T}} := E_{\mathcal{T}} \cup \{e_m\}$
$\leq_{\mathcal{T}} := \leq \cap (E_{\mathcal{T}} \times E_{\mathcal{T}})$
$\#_{\mathcal{T}} := \# \cap (E_{\mathcal{T}} \times E_{\mathcal{T}})$
$\lambda_{\mathcal{T}} := \lambda_{|E_{\mathcal{T}}}$
**return** $\mathcal{T} = (E_{\mathcal{T}}, \leq_{\mathcal{T}}, \#_{\mathcal{T}}, \lambda_{\mathcal{T}})$

*given as the inputs of Algorithm 3, the obtained test case is $\mathcal{T}_8$ from Figure 5.6. Event $e_9$ is not added since it would add immediate conflict to the test case, while events $e_{10}, e_{11}$ are not part of the test case since an event in their past ($e_9$) was removed.*

### 5.2.3 IICS Set

In order to cover all the branches of the event structure, the algorithm must be run several times with different conflict resolution schemes, to obtain a test suite that represents every possible event in at least one test case. Instead of running the algorithm with all possible interleavings, the collection of linearizations that are used needs only to consider all resolutions of immediate input conflict, i.e. there must be a pair of linearizations that reverses the order in a given immediate input conflict.

**Definition** 5.6 (IICS Set). *Fix $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$, 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 $e_1, e_2 \in E^{\mathcal{I}n}$ such that $e_1 \#^r e_2$, there exist $\mathcal{R}_1, \mathcal{R}_2 \in \mathcal{L}$ such that $e_1 \mathcal{R}_1 e_2$ and $e_2 \mathcal{R}_2 e_1$.*

An iics set allows to cover all the branches of an event structure using Algorithm 3.

**Proposition** 4. *Let $\mathcal{L}$ be an iics set for $\mathcal{E} = (E, \leq, \#, \lambda)$, and $T$ the test suite obtained using Algorithm 3 with $\mathcal{L}$. Then every event $e \in E$ is represented by at least one test case $\mathcal{T} \in T$.*

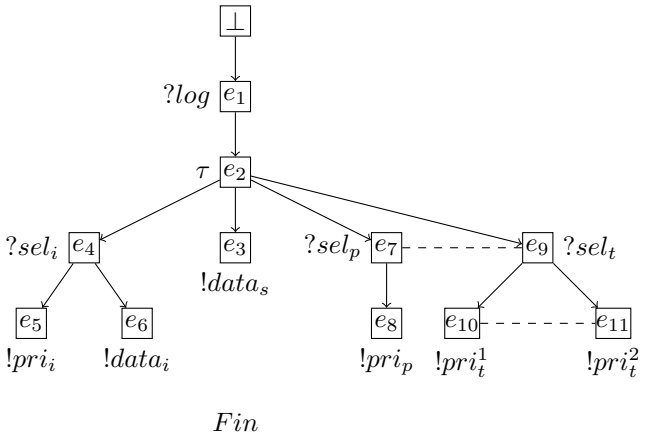



Figure 5.8: A finite prefix of the unfolding $\mathcal{E}_{ag}$.

*Proof.* Let $T$ be the test suite obtained by the algorithm and $\mathcal{L}$, and suppose $e$ is not represented by any test case in $T$. We have then that for every $\mathcal{T} \in T$ either *(i)* $e \in E^{\mathcal{I}n}$ and $(\{e\} \times E^{\mathcal{I}n}_{\mathcal{T}}) \cap \#^r \neq \emptyset$ or *(ii)* $\langle e \rangle \not\subseteq E_{\mathcal{T}}$. If *(i)*, we have that there exists $e' \in E^{\mathcal{I}n}_{\mathcal{T}}$ such that $e \#^r e'$ and $e' \mathcal{R}_1 e$ (where $\mathcal{R}_1$ is the linearization used to build $\mathcal{T}$). By Definition 5.6, 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 $\mathcal{T}' \in T$ such that $e$ is represented by $\mathcal{T}'$, which leads to a contradiction. If *(ii)*, then there exists $e' \in \langle e \rangle$ such that $(\{e'\} \times E^{\mathcal{I}n}_{\mathcal{T}}) \cap \#^r \neq \emptyset$ and the analysis is analogous to the one in *(i)*. □

Example 5.11 (IICS Set). *Consider the event structure $Fin$ of Figure 5.8 and linearizations $\mathcal{R}_1 = \bot \cdot e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_7 \cdot e_8 \cdot e_9 \cdot e_{10} \cdot e_{11}$ and $\mathcal{R}_2 = \bot \cdot e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_9 \cdot e_{10} \cdot e_{11} \cdot e_7 \cdot e_8$. The only immediate conflict between inputs in $Fin$ is between $e_7$ and $e_9$. The two linearizations reverse the order between these events and therefore they form an iics set. Algorithm 3 constructs $\mathcal{T}_9$ from Figure 5.7 when $\mathcal{R}_2$ is used; the events from $Fin$ that were missing in $\mathcal{T}_8$, i.e. $e_9, e_{10}, e_{11}$, are part of $\mathcal{T}_9$ and therefore the whole prefix is covered.*

Consider the unfolding $\mathcal{S}$ of the specification of the system and the set of all its possible prefixes which is denoted by $\mathcal{PREF}(\mathcal{S})$. Algorithm 3 is general enough to produce a complete test suite.

**Theorem** 8. *Let $\mathcal{S}$ be the unfolding of the specification. From $\mathcal{PREF}(\mathcal{S})$ and a given iics set $\mathcal{L}$ for $\mathcal{S}$, Algorithm 3 yields a complete test suite $T$.*

*Proof. Soundness:* By Theorem 6 we need to prove that: *(i)* the traces of every test case are traces of the specification; *(ii)* the outputs following a trace of the test are at least those specified; *(iii)* any concurrent complete set of possible input in the test case is concurrent complete in the specification. *(i)* Trace inclusion is immediate since the algorithm only removes events generating immediate conflict. *(ii)* For a test $\mathcal{T}$ and a trace $\omega \in \text{traces}_r(\mathcal{T})$, if an output in

$\text{out}_r(\mathcal{S}\ \textbf{after}_r\ \omega)$ is not in $\text{out}_r(\mathcal{T}\ \textbf{after}_r\ \omega)$, it means either that it is in conflict with an input in $\mathcal{T}$, which is impossible by Assumption 2, or that its past is not already in $\mathcal{T}$, which is impossible since $\omega$ is a trace of $\mathcal{T}$. *(iii)* 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 obtained event structure 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 7 we need to prove: *(i)* every trace is represented in at least one test case; *(ii)* the test case does not produce outputs that are not specified; *(iii)* concurrent complete set of inputs of the specification remain as concurrent complete sets in the test case. *(i)* Clearly, for all $\omega \in \text{traces}_r(\mathcal{S})$ there exists at least one prefix $\mathcal{P} \in \mathcal{PREF}(\mathcal{S})$ such that $\omega \in \text{traces}_r(\mathcal{P})$. By Proposition 4 we can find $\mathcal{R} \in \mathcal{L}$ such that this trace remains in the test case obtained by the algorithm. *(ii)-(iii)* The inclusion of outputs and preservation of concurrent complete sets is immediate since the algorithm does not add events.

□

### 5.2.4 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 unfolding, we construct one test case for each linearization in $\mathcal{L}$. We present an upper bound for the size of $\mathcal{L}$ and discuss how to improve on it.

Example 5.12 (Size of a IICS Set). *Consider the event structure Fin of Figure 5.8 and linearizations*

$$\begin{aligned}\mathcal{R}_1 &= \bot \cdot e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_7 \cdot e_8 \cdot e_9 \cdot e_{10} \cdot e_{11}\\ \mathcal{R}_2 &= \bot \cdot e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_9 \cdot e_{10} \cdot e_{11} \cdot e_7 \cdot e_8\\ \mathcal{R}_3 &= \bot \cdot e_1 \cdot e_2 \cdot e_3 \cdot e_4 \cdot e_5 \cdot e_6 \cdot e_9 \cdot e_7 \cdot e_8 \cdot e_{10} \cdot e_{11}\end{aligned}$$

*Algorithm 3 produces test cases $\mathcal{T}_8, \mathcal{T}_9$ using linearizations $\mathcal{R}_1$ and $\mathcal{R}_2$ respectively. We can easily see that some events commute between $\mathcal{R}_2$ and $\mathcal{R}_3$, however, the algorithm constructs $\mathcal{T}_9$ whichever of them we use.*

The example above shows that some commutations of events in the linearization produce different test cases, while others do not. The concept of partial commutation was introduced by Mazurkiewicz [DR95] 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 $\Sigma$ be a finite alphabet and $I \subseteq \Sigma \times \Sigma$ a symmetric and irreflexive relation called independence or commutation. The relation $I$ induces an equivalence relation $\equiv_I$ over $\Sigma^*$. Two words $x$ and $y$ are equivalent, denoted by $x \equiv_I y$, 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 pair of letters $(a_i, b_i) \in I$ satisfying

$$z_i = z_i' a_i b_i z_i'' \text{ and } z_{i+1} = z_i' b_i a_i z_i''$$

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 5.13 (Mazurkiewicz Traces). *Consider* $\Sigma = \{a, b, c, d\}$ *and the independence relation* $I = \{(a, d)(d, a)(b, c)(c, d)\}$*, 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 [RS97] that every (Mazurkiewicz's) trace has a unique normal form (every trace in the equivalence class has the same one). Different linearizations having the same normal form, construct the same test case.

We have seen that the order between concurrent events or output events in immediate conflict does not change the test cases constructed by Algorithm 3, but immediate conflict between inputs and causality does. We propose the following independence relation:

$$I_{\mathcal{S}} \triangleq (E \times E) \backslash (\leq \cup ((E^{\mathcal{I}n} \times E^{\mathcal{I}n}) \cap \#^r))$$

For constructing a test case, we need to 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 $\equiv_{I_s}$.

**Theorem** 9. *Let* $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ *and* $\mathcal{R}_2 \in [\mathcal{R}_1]_{I_S}$ *for some linearization* $\mathcal{R}_1$ *of* $\leq$*. Algorithm 3 generates the same test cases with both* $\mathcal{R}_1$ *and* $\mathcal{R}_2$*.*

*Proof.* Let $\mathcal{T}_1 = (E_1, \leq_1, \#_1, \lambda_1)$ and $\mathcal{T}_2 = (E_2, \leq_2, \#_2, \lambda_2)$ the test cases obtained by linearizations $\mathcal{R}_1, \mathcal{R}_2$ respectively and suppose $\exists e \in E_1 : e \notin E_2$. Since $e$ was removed from $\mathcal{T}_2$, then either: *(i)* $\exists e' \in E_2^{\mathcal{I}n} : e'\mathcal{R}_2 e \wedge e \ \#^r \ e'$; or *(ii)* $\exists e' \in \langle e \rangle : e' \notin E_2$. If *(i)*, Assumption 2 implies $e \in E_1^{\mathcal{I}n}$, thus $(e, e') \notin I$. Since $\mathcal{R}_1 \equiv_{I_S} \mathcal{R}_2$, $\mathcal{R}_1$ cannot reverse the order between $e$ and $e'$ and $e'\mathcal{R}_1 e$. $e' \in E_2^{\mathcal{I}n}$ implies that there is not a smaller event (w.r.t $\mathcal{R}_2$) that prevents it to be part of the test case, and this is also the case in $\mathcal{R}_1$ because they are equivalent. We can conclude that $e' \in E_1^{\mathcal{I}n}$ and since $e \ \#^r \ e'$ and $e'\mathcal{R}_1 e$, it follows that $e \notin E_1$, which contradicts the hypothesis. If *(ii)*, then $\exists e'' \in E_2^{\mathcal{I}n} : e''\mathcal{R}_2 e' \wedge e'' \ \#^r \ e'$ and the analysis is similar to *(i)*. □

**Corollary** 1. *Let* $\mathcal{K} = |(E^{\mathcal{I}n} \times E^{\mathcal{I}n}) \cap \#^r|$*, then Algorithm 3 needs to be run only* $2^{\mathcal{K}}$ *times to obtain a test suite that covers the prefix.*

*Proof.* Since $\mathcal{K}$ is the number of equivalence classes for all the possible linearizations of $\leq$, the result is immediate using Theorem 9. □

The corollary above gives an upper bound in the number of test cases needed, however there may still be some redundancy in the test suite.

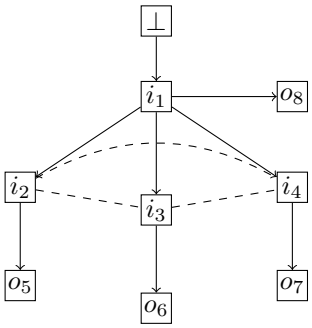



Figure 5.9: An event structure with three inputs in immediate conflict

Example 5.14 (Redundant Test Cases in a Test Suite). *Consider the event structure from Figure 5.9 with* $i_2, i_3, i_4 \in E^{\mathcal{I}n}$ *and linearizations of the causality relation* $\mathcal{R} = \bot \cdot i_1 \cdot i_2 \cdot o_8 \cdot i_3 \cdot o_6 \cdot i_4 \cdot o_5 \cdot o_7$ *and* $\mathcal{R}' = \bot \cdot i_1 \cdot o_8 \cdot i_2 \cdot o_5 \cdot i_3 \cdot o_6 \cdot i_4 \cdot o_7$. *The normal form of both* $\mathcal{R}$ *and* $\mathcal{R}'$ *is* $(\bot)(i_1)(i_2)(i_3)(i_4)(o_5o_6o_7o_8)$, *meaning that the order of* $o_5, o_6, o_7, o_8$ *is not really important for constructing the test case. For any linearization of the causality relation, its normal form is one of the followings:*

$$\mathcal{R}_1 = (\bot)(i_1)(i_2)(i_3)(i_4)(o_5o_6o_7o_8) \quad \mathcal{R}_2 = (\bot)(i_1)(i_2)(i_4)(i_3)(o_5o_6o_7o_8)$$
$$\mathcal{R}_3 = (\bot)(i_1)(i_3)(i_2)(i_4)(o_5o_6o_7o_8) \quad \mathcal{R}_4 = (\bot)(i_1)(i_3)(i_4)(i_2)(o_5o_6o_7o_8)$$
$$\mathcal{R}_5 = (\bot)(i_1)(i_4)(i_2)(i_3)(o_5o_6o_7o_8) \quad \mathcal{R}_6 = (\bot)(i_1)(i_4)(i_3)(i_2)(o_5o_6o_7o_8)$$

*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. Linearizations* $\mathcal{R}_1, \mathcal{R}_3$ *and* $\mathcal{R}_5$ *are sufficient to cover the specification.*

Next section proposes a SAT encoding of test cases that not only removes this redundancy in the test suite construction, but it also takes care of uncontrollably problems.

### 5.2.5 SAT Encoding of Test Cases

Test cases cannot be constructed directly during unfolding time as some needed information may not be available in the current unfolding prefix.

Example 5.15 (Test Case Generation During the Unfolding Procedure). *Suppose one wants to unfold a system which complete unfolding is given by Figure 5.10 and suppose the current prefix only contains events* $i_1$ *and* $o_1$. *When we compute the possible extensions, we obtain* $i_2, i_3$ *and we already know they are in immediate conflict and only one of them can be part of the test case. If we only have* $i_1$ *in the prefix, the possible extensions are* $i_2, o_1$. *If we select* $i_2$, *this prohibits*

*adding $i_3$ in the future as it adds immediate conflict between inputs, however, this information is not available at the moment of selecting between $i_2$ or $o_1$.*

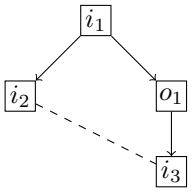



Figure 5.10: Test case generation during the unfolding procedure.

In order to avoid this problem and the redundancy mentioned in Example 5.14, we propose to use a SAT encoding which also solves controllability problems. We use a SAT variable $\varphi_e$ for each event $e$, thus we need an event structure with a finite number of events (see Section 5.3 for information about how to obtain a finite prefix of the unfolding). We construct a SAT formula where each solution represents a test case, i.e. a solution assigning 1 to variable $\varphi_e$ means that event $e$ belongs to the test case, while assignment 0 means that it does not.

As test cases preserve causality, whenever the condition of an event is true, the condition of every event in its past should also be true. In addition, for each pair of immediate conflict between inputs, exactly one of them can belong to the test case. We intend the test suite to cover the whole prefix, therefore the test cases should be maximal in the sense that adding any event should violate the causal precedence or non immediate conflict between inputs properties. Since immediate conflicts between inputs and outputs are not accepted in the specification, an output of the specification does not belong to the test case only if one of the events in its past does not belong either. The same is also true for internal events. In the case of an input event, if it does not belong to the test case, it should either violate the causal precedence property, or introduce immediate conflicts between inputs forbidden by Definition 5.1. These restrictions are captured by the conjunction of the following SAT formulas for each event.

$$\forall e \in E : \bigwedge_{f \leq e} \varphi_e \Rightarrow \varphi_f \tag{5.1}$$

$$\forall e \in E^{\mathcal{I}n} : \bigwedge_{f \#^r e} \neg\varphi_e \vee \neg\varphi_f \tag{5.2}$$

$$\forall e \in (E^{\mathcal{O}ut} \cup E^{\tau}) : \neg\varphi_e \Rightarrow \bigwedge_{f \leq e} \neg\varphi_f \tag{5.3}$$

$$\forall e \in E^{\mathcal{I}n} : \neg\varphi_e \Rightarrow (\bigvee_{f \leq e} \neg\varphi_f \vee \bigvee_{f \#^r e} \varphi_f) \tag{5.4}$$

Example 5.16 (SAT Encoding of the Travel Agency). *Consider the event structure of Figure 5.11. The SAT formula contains the following constrains for*

*events* $e_9$ *and* $e_{10}$

$$(\varphi_{e_9} \Rightarrow (\varphi_{e_2} \wedge \varphi_{e_1} \wedge \varphi_{\perp})) \wedge (\neg\varphi_{e_7} \vee \neg\varphi_{e_9}) \wedge (\neg\varphi_{e_9} \Rightarrow (\neg\varphi_{e_2} \vee \neg\varphi_{e_1} \vee \neg\varphi_{\perp} \vee \varphi_{e_7}))$$

$$(\varphi_{e_{10}} \Rightarrow (\varphi_{e_9} \wedge \varphi_{e_2} \wedge \varphi_{e_1} \wedge \varphi_{\perp})) \wedge (\neg\varphi_{e_{10}} \Rightarrow (\neg\varphi_{e_9} \vee \neg\varphi_{e_2} \vee \neg\varphi_{e_1} \vee \neg\varphi_{\perp}))$$

*The complete SAT formula reduces to*

$$\varphi_{\perp} \wedge \varphi_{e_1} \wedge \varphi_{e_2} \wedge \varphi_{e_3} \wedge \varphi_{e_4} \wedge \varphi_{e_5} \wedge \varphi_{e_6} \wedge ((\varphi_{e_7} \wedge \varphi_{e_8} \wedge \neg\varphi_{e_9} \wedge \neg\varphi_{e_{10}} \wedge \neg\varphi_{e_{11}}) \vee (\neg\varphi_{e_7} \wedge \neg\varphi_{e_8} \wedge \varphi_{e_9} \wedge \varphi_{e_{10}} \wedge \varphi_{e_{11}}))$$

*which is satisfiable by the two assignments*

$$\varphi_{\perp} = \varphi_{e_1} = \varphi_{e_2} = \varphi_{e_3} = \varphi_{e_4} = \varphi_{e_5} = \varphi_{e_6} = \varphi_{e_9} = \varphi_{e_{10}} = \varphi_{e_{11}} = 1,$$
$$\varphi_{e_7} = \varphi_{e_8} = 0$$

$$\varphi_{\perp} = \varphi_{e_1} = \varphi_{e_2} = \varphi_{e_3} = \varphi_{e_4} = \varphi_{e_5} = \varphi_{e_6} = \varphi_{e_7} = \varphi_{e_8} = 1,$$
$$\varphi_{e_9} = \varphi_{e_{10}} = \varphi_{e_{11}} = 0$$

*which represent the test cases* $\mathcal{T}_8$ *and* $\mathcal{T}_9$ *respectively.*

Example 5.17 (Avoiding Redundancy by SAT Encoding). *Consider the event structure of Figure 5.9. The constraints generated by Equation 5.2 imposes that exactly one between* $\varphi_{e_2}, \varphi_{e_3}, \varphi_{e_4}$ *is equal to* 1*. The SAT formula has three solutions representing the only necessary test cases, avoiding the redundancy seen in Example 5.14.*

We will use this SAT encoding in Appendix B for the generation of test cases for several examples and show its efficiency when partial order semantics are used.

## 5.3 Test Selection

While sufficient conditions for soundness and exhaustiveness of test suites have been given, exhaustive test suites are usually infinite or contain infinite test cases, while 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), while preserving soundness.

In the **ioco** framework and its extensions, the selection of test cases is achieved by different methods. Tests can be built in a randomized way from a canonical tester, which is a completion of the specification representing all the authorized and forbidden behaviors [Tre96b]. Closer to practice is the selection of tests according to test purposes, which represent a set of behaviors one wants to test [JJ05]. Another method, used for symbolic transition systems for instance, is to unfold the specification until a certain testing criterion is fulfilled, and then to build a test suite covering this unfolding. Criteria for stopping the unfolding can be a given depth or state inclusion for instance [GGRT06].

### 5.3.1 Coverage Criteria Based on Cut-off Events

We propose testing criteria based on the structure of the specification. This criteria includes all-transitions and all-states criteria [GS04] where each transition or state of the specification must be covered by at least one test case. Other criteria include covering every loop or cycle a fixed number of times or every possible path, however the latter is usually impossible to achieve in practice.

The entire behavior of a Petri net is captured by its unfolding, but this unfolding is usually infinite both in depth and breadth. Infinite depth needs infinite test cases to cover every path while infinite breadth needs an infinite set of (possible finite) test cases. There are several different methods of truncating an unfolding and then avoiding both infinite test cases and test suites. The differences are related to the kind of information about the original unfolding one wants to preserve in the prefix. The algorithm for constructing a finite prefix [McM95] depends on the notion of cut-off event: how long the net is unfolded. Our aim is to use such a prefix to build test cases, therefore obtaining a finite prefix can be seen as defining a testing criterion.

Theorem 6 shows that if the information about the produced outputs (and quiescence) is preserved in the test cases, we can prove the soundness of the test suite. Hence we aim at truncating the unfolding following a specific criterion, while preserving information about outputs and quiescence. In order to preserve this information, we follow [GGRT06] and modify the finite prefix algorithm [McM95] adding all the outputs from the unfolding that the prefix enables. As there exists no cycles of outputs in the original net, this procedure terminates, yielding a finite prefix. The procedure to compute the quiescent closure of a finite prefix (denoted by $Fin^{\Theta}$) is described by Algorithm 4.

The algorithm takes as an input a net and a notion of cut-off or termination (when to cut a branch). It starts from an empty prefix (as in the case of the unfolding algorithm) and at each iteration computes the possible extensions of the current prefix. If a possible extension is such that it is not in the future of a cut-off or terminal event, the event is added to the prefix, if not, it is removed from the possible extensions, i.e. events in the future of a cut-off event are not added in the first unfolding loop. Once the net is unfolded into a finite prefix, for every maximal (w.r.t set inclusion) configuration of the prefix enabling outputs events, those events are added to the prefix. The latter assures that the produced outputs of the prefix and the unfolding coincide and it is used to show that the test suite obtained from such prefix is sound.

Algorithm 4 is parametric on the cutting criterion: if we change the notion of cutting event, the finite prefix obtained is different. As in [JM99] where the complete test graph contains all test cases corresponding to the test purpose, each of these prefixes contain all the test cases corresponding to its corresponding testing criterion. We propose three cut-off notions corresponding to different testing criteria.

**All Paths of Length $n$ Criterion.** The first cut-off notion we present depends on the height of an event, defined as the length of the longest causality

**Algorithm 4** Quiescent closure of a finite prefix

**Require:** A Petri net $\mathcal{N} = (P, T, F, \lambda, M_0)$ where $M_0 = \{p_1, \ldots, p_n\}$, and a cut-off predicate on events

**Ensure:** A finite prefix $Fin^\Theta$ of the unfolding $\mathcal{U}$ of net $\mathcal{N}$ such that $\forall \omega \in \text{traces}(Fin^\Theta) : \text{out}_r(Fin^\Theta \textbf{ after}_r\ \omega) = \text{out}_r(\mathcal{U} \textbf{ after}_r\ \omega)$

1: $Fin^\Theta := \{(\bot, \emptyset), (p_0, \bot), \ldots, (p_n, \bot)\}$
2: $pe := PE(Fin^\Theta)$
3: *cut-off* $:= \emptyset$
4: **while** $pe \neq \emptyset$ **do**
5: choose an event $e = (t, C)$ in $pe$
6: **if** $[e] \cap$ *cut-off* $= \emptyset$ **then**
7: append to $Fin^\Theta$ event $e$ and a condition $(p, e)$ for every place $p$ in $t^\bullet$
8: $pe := PE(Fin^\Theta)$;
9: **if** $e$ is a cut-off event of $Fin^\Theta$ **then**
10: *cut-off* $:=$ *cut-off* $\cup \{e\}$
11: **else**
12: $pe := pe \backslash \{e\}$
13: $pe := PE(Fin^\Theta)$
14: **while** $\exists (t, C) \in pe : t \in T^{\mathcal{O}ut}$ **do**
15: choose an event $e = (t, C)$ in $pe$ such that $t \in T^{\mathcal{O}ut}$
16: append to $Fin^\Theta$ the event $e$ and a condition $(p, e)$ for every place $p$ in $t^\bullet$
17: $pe := PE(Fin^\Theta)$;
**return** $Fin^\Theta$

chain containing this event. It defines a selection criterion similar to the criterion "all paths of length $n$" defined in [GGRT06].

**Definition** 5.7 (Height of an Event). *For a branching process $Fin$, define the height of an event $e$ in $Fin$ recursively by*

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

The following criterion generates a prefix where the height of any event is at most $n$.

**Definition** 5.8 (n-Cut-off Events). *Let $Fin$ be a branching process. An event $e$ is an $n$-cut-off event iff $\mathcal{H}(e) = n$.*

Example 5.18 (All Paths of Length $n$ Criterion). *Figure 5.11 shows a finite prefix of the travel agency unfolding $\mathcal{E}_{ag}$ using an "all path of length 3" criterion. The events in $S_1$ are those marked as cut-off (their height is 3). During the first while condition of Algorithm 4 (lines 4-12), only events $\bot, e_1, e_2, e_3, e_4, e_7, e_9$ are added. In this prefix, the maximal configuration $\{\bot, e_1, e_2, e_3, e_4, e_7\}$ enables outputs $!pri_i, !data_i$ and $!pri_p$. The second while condition (lines 13-17) builds the quiescent closure by adding output events in $S_2$.*

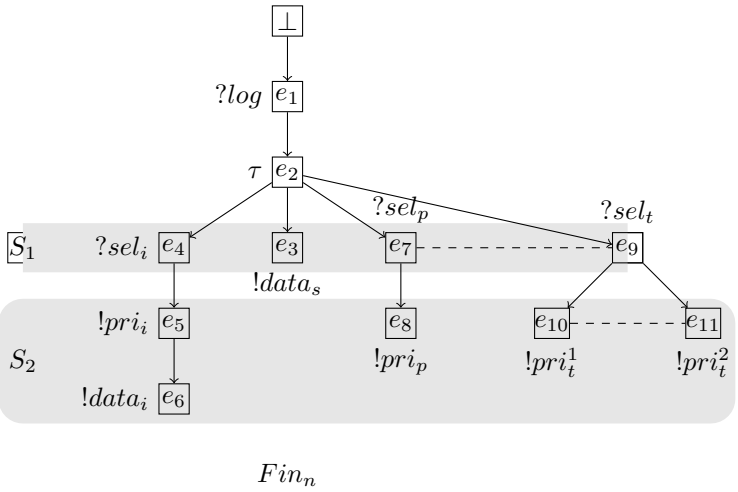



Figure 5.11: All paths of length $n$ criterion.

This criterion allows us to build test cases that cover all paths of height $n$, however, the pertinent $n$ to be chosen is up to the tester. Notice that the behavior of the system consists usually of infinite traces, however, in practice, these long traces can be considered as a sequence of (finite) "basic" behaviors. For example, the travel agency offers few basic behaviors: *(i)* logging in; *(ii)* selection of insurance; and *(iii)* selection of tickets. Any "complex" behavior of the agency is built from such basic behaviors. The longest length of these basic behaviors can be chosen as a pertinent length to unfold.

**Inclusion Criterion.** From the observation that a specification generally describes a set of basic behaviors that eventually repeat themselves, another natural criterion consists in covering the cycles of the specification. We define a criterion allowing to cover each basic behavior at least once, using a proper notion of complete prefix [McM95].

**Definition** 5.9 (Complete Prefixes). *A branching process $\beta$ of an Petri net $\mathcal{N}$ is complete if for every reachable marking $M$ there exists a configuration $C$ in $\beta$ such that:*

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

The first condition in Definition 5.9 implies that every complete prefix covers every state of the system, i.e. it assures a all-states coverage; while the second condition assures all-transitions coverage.

Note that the completeness of a prefix does not imply that the information about outputs and quiescence is preserved, so Algorithm 4 still needs to add outputs to build its quiescent closure.

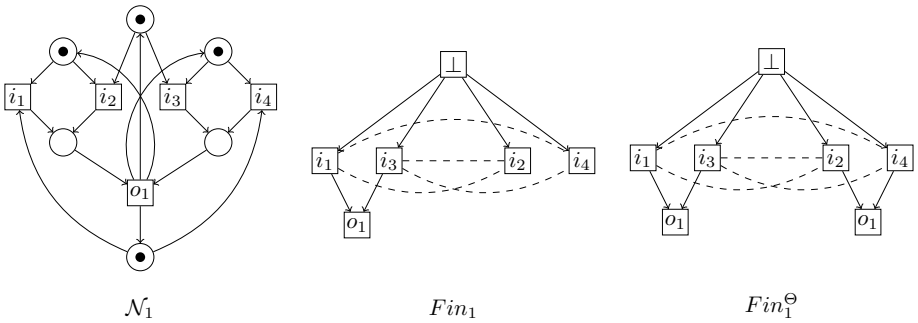



Figure 5.12: A Petri net $\mathcal{N}$, a complete finite prefix $Fin$ and its quiescent closure $Fin^{\Theta}$.

Example 5.19 (Complete Prefixes and Produced Outputs). *The prefix $Fin_1$ of the unfolding of net $\mathcal{N}_1$ from Figure 5.12 is complete, however the expected outputs are not part of the prefix. Output $o_1$ is produced by $\mathcal{N}_1$ after $i_2$ and $i_4$, but this is not the case in $Fin_1$, i.e. $o_1 \notin out_r(Fin_1 \ \textbf{\textit{after}}_r \ (i_2 \ \mathbf{co} \ i_4)) = \{\delta\}$. The quiescent closure $Fin_1^{\Theta}$ adds all the necessary outputs.*

The following cut-off notion corresponds to the inclusion criterion presented in [GGRT06]; the unfolding can be truncated when the current event being unfolded generates a marking that was already seen in the current branch.

**Definition** 5.10 (Inclusion Cut-off Events). *Let $Fin$ be a branching process. An event $e$ is an inclusion cut-off event iff $Fin$ contains an event $e' \leq e$ such that $Mark([e']) = Mark([e])$.*

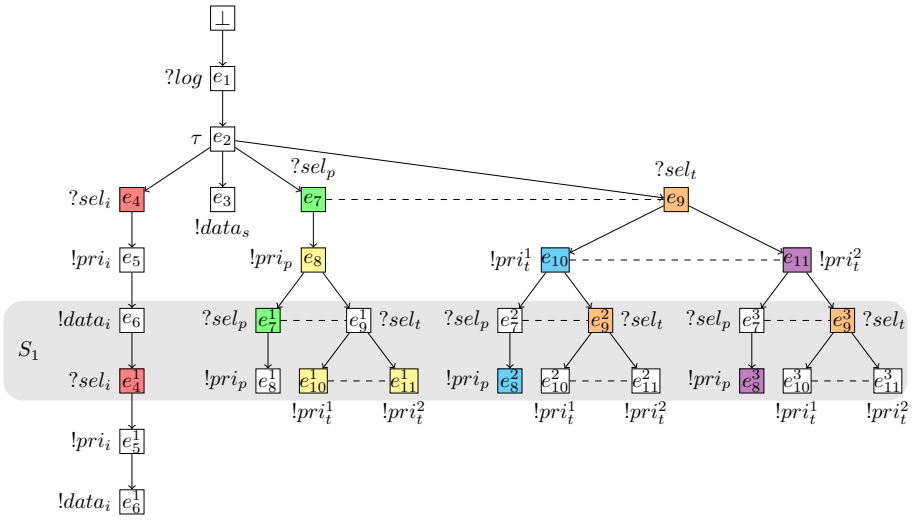



Figure 5.13: Inclusion criterion.

Example 5.20 (Inclusion Criterion). *Figure 5.13 shows the output closure of the finite prefix obtained using the inclusion criterion for the net $\mathcal{N}_{ag}$. The colored events in $S_1$, i.e. $e_4^1, e_7^1, e_{10}^1, e_{11}^1, e_8^2, e_9^2, e_8^3, e_9^3$, are marked as cut-off. Colors relate cut-off events with the corresponding smaller event generating the same marking, for example $e_4 \leq e_4^1$ and $Mark([e_4]) = Mark([e_4^1])$. As in the case of all path n criterion, some outputs need to be added to the prefix: $e_7^1$ is marked as a cut-off, but the output event $e_8^1$ is also part of the prefix.*

The finite prefix obtained using the inclusion cut-off notion is complete and therefore includes all-transitions and all-states coverage. However as it is shown in [ERV02], it is not necessarily the minimal complete prefix.

Example 5.21 (Minimal Complete Prefixes and Testing). *Figure 5.14 presents a Petri net $\mathcal{N}_2$ together with the finite prefix $Fin_2$ obtained using the inclusion criterion. Esparza et al. [ERV02] proposed an improvement to the finite complete prefix algorithm, to obtain a (potentially) smaller prefix, as it is the case of $Fin'_2$. Smaller complete prefixes can lead to exponential reductions in verification techniques, however, in testing, smaller prefixes reduce the testing power of the test suite. Consider $\mathcal{N}_2$ as the specification of the system and an implementation which accepts $?i_2$, but after, it refuses $?i_3$. Clearly this implementation is non conformant w.r.t any of our conformance relations (where we do not assume input-enabledness). The test suite obtained from prefix $Fin'_2$ after solving immediate conflict between inputs is $\{?i_1 \cdot ?i_3, ?i_1 \cdot ?i_4, ?i_2\}$. The implementation does not fail the test suite as the trace $?i_2 \cdot ?i_3$ is never tested and non conformance is not detected. The test suite obtained from $Fin_2$ contains $?i_2 \cdot ?i_3$ and therefore it detects the refused input.*

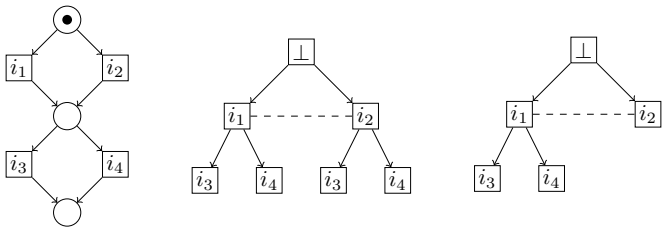



Figure 5.14: Minimal complete prefixes and testing.

***k*-inclusion Criterion.** A natural extension of the previous criterion consists in traversing each basic behavior a fixed number of times. We present below the $k$-inclusion criterion which together with Algorithm 4 leads to a complete prefix representing each basic behavior at least $k$ times in each branch and preserving outputs and quiescence.

**Definition** 5.11 (k-Inclusion Cut-off Events). *Let $Fin$ be a branching process. An event $e$ is a $k$-inclusion cut-off event iff $Fin$ contains a family of $k$ events $\{e_i\}_{i \leq k}$ such that $e_i \leq e_{i+1} \leq e$ and $Mark([e_i]) = Mark([e])$ for $0 \leq i < k$.*

Obviously a 1-inclusion cut-off event is an inclusion cut-off event in the sense of Definition 5.10. As the $k$-inclusion criterion extends the prefix obtained by the inclusion criterion, clearly this prefix is also complete and therefore assures all-transitions and all-states coverage.

### 5.3.2 Soundness of the Test Suite

We have shown how to obtain a finite prefix of the unfolding that not only reduces the size of both test cases (avoiding infinite paths) and the test suite (avoiding infinite branchings), but it also covers a significant part of the specification behavior. The following result is central and will help proving soundness of the test suite constructed from the prefix.

**Theorem** 10. *Let $\mathcal{N}$ be a net, $\mathcal{U}$ its unfolding and $Fin^{\Theta}$ the quiescent closure of one of its prefixes obtained either by the $k$-inclusion[1] or the all paths of length $n$ criterion and Algorithm 4, then*

1. $traces_r(Fin^{\Theta}) \subseteq traces_r(\mathcal{U})$
2. $\forall \omega \in traces_r(Fin^{\Theta}) : out_r(Fin^{\Theta} \ \textbf{\textit{after}}_r \ \omega) = out_r(\mathcal{U} \ \textbf{\textit{after}}_r \ \omega)$

*Proof.* 1) is immediate since $Fin^{\Theta}$ is a prefix of $\mathcal{U}$. 2) The second loop of Algorithm 4 adds every output which is enabled by the current prefix, therefore it only removes outputs if an input in their past was also removed. As only the outputs produced after the traces of $Fin^{\Theta}$ are considered, the result is immediate. □

We can now prove the main result of this section: the test suites constructed based on the $k$-inclusion or the all paths of length $n$ criterion are sound.

**Theorem** 11. *Let $\mathcal{U}$ be the unfolding of the specification of a system and $Fin^{\Theta}$ the quiescent closure of one of its prefixes obtained by Algorithm 4. Any test suite constructed by Algorithm 3 or the SAT encoding and $Fin^{\Theta}$ is sound w.r.t* ***wsc-ioco****.*

*Proof.* By Theorem 6, we need to prove that any trace of a test case $\mathcal{T}$ is a trace of the unfolding (which is trivial as $\mathcal{T}$ is a prefix of $Fin^{\Theta}$ and therefore of $\mathcal{U}$) and that outputs and quiescence produced after any trace of such a test are preserved. The events of $Fin^{\Theta}$ that are added to $\mathcal{T}$ by Algorithm 3 or the SAT encoding are those whose past is already in $\mathcal{T}$ and which are not in immediate conflict with its inputs. An output cannot be in immediate conflict with an input by Assumption 2, so all the outputs from $Fin^{\Theta}$ whose past is already in $\mathcal{T}$ are added. This implies that all the outputs of $Fin^{\Theta}$ after a trace of the test case are preserved and by Theorem 10 we have $\forall \omega \in \text{traces}_r(\mathcal{T}) : \text{out}_r(\mathcal{T} \ \textbf{after}_r \ \omega) = \text{out}_r(Fin^{\Theta} \ \textbf{after}_r \ \omega) = \text{out}_r(\mathcal{U} \ \textbf{after}_r \ \omega)$. □

[1] This criterion includes the inclusion criterion.

### 5.3.3 Comparing Different Criteria

Example 5.21 shows that different prefixes may lead to smaller test suites, but which have less testing power. We need therefore a way to compare the testing power between prefixes. We can follow the $k$-inclusion criterion and consider the number of times a marking is covered as a measure for the quality of the test suite, however this does not consider the "cost" of executing such test suite. To measure the quality of a prefix, we propose to balance between the testing power and the size of the prefix.

We define the coverage of a configuration as the number of times its corresponding marking is represented in any proper subset:

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

In order to take into account the "cost" of executing a larger prefix to increase the coverage of its configuration, we also consider in the quality of a configuration the number of events of the prefix:

$$\mathcal{Q}(C) \triangleq \frac{\mathcal{C}ov(C)^2}{|Fin|}$$

Let $\Omega(\mathcal{E})$ denote the set of maximal (w.r.t set inclusion) configurations of $\mathcal{E}$. The quality of a prefix is defined as the smallest quality of its maximal configurations.

**Definition** 5.12 (Quality of a Prefix). *Let $Fin$ be a finite prefix of the unfolding, we define the quality of $Fin$ as*

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

Example 5.22 (Testing Quality of a Prefix). *Consider the net $\mathcal{N}_3$ from Figure 5.15 and the two finite prefixes of its unfolding $Cov_1$ and $Cov_2$ obtained by the 1-inclusion and 2-inclusion criteria respectively. The coverage of each maximal configuration in $Cov_1$ is equal to 1 and then $\mathcal{Q}(Cov_1)$ is $1/5 = 0.2$. In order to obtain a coverage equal to 2, we need to add four events to $Cov_2$, thus $\mathcal{Q}(Cov_2) = 2^2/13 \approx 0.3$. The number of events that need to be added to obtain a coverage of 3 in maximal configurations is too big (48 new events) and reduces the quality of the prefix, i.e. $\mathcal{Q}(Cov_3) = 3^2/61 \approx 0.14$, where $Cov_3$ is the prefix obtained by a 3-inclusion criterion.*

The fact that a large number of events needs to be added in the unfolding to increase the coverability of maximal configurations is one of the disadvantages of unfoldings. There exist other ways to unfold the net to avoid such kind of explosion (for example merged processes [KKKV06]). However, this condensed way of representing the unfolding of the system comes at a price, for example, identifying its configuration its not straightforward as merged processes usually contain cycles.

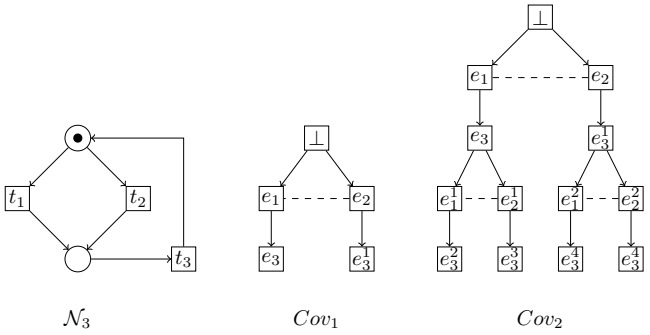



Figure 5.15: Testing quality of a prefix.

## 5.4 Conclusion

This chapter introduced a complete testing framework for concurrent systems based on a centralized testing architecture, i.e. global control and observation of the system is assumed.

We introduced the notion of global test cases which allow concurrency in the tests. However, in practice, such global test cases are not meant to be actually executed globally, they would rather be projected onto the different processes to be executed locally, in order to make the observation of concurrency possible. Our approach here is to study the testing problem from a centralized point of view, as a basis to the distributed testing problem: the global conformance relation (**co-ioco**) is the relation we want to test with local control and observation (see Chapter 6), and the global test cases are the basis for the construction of local tests. We showed that in general, global test cases are not only smaller that local test cases, but also than the number of global test cases in a test suite is usually smaller than the number of local test cases. This is due to the fact that interpreting concurrency by interleavings introduces controllability problems that need to be solved by the test case generation algorithm.

Since global test cases are not meant to be executed globally, we did not define a notion of test execution, but we rather consider the situations where the interaction between the tester and the implementation is not possible in order to define verdicts that allow to decide conformance. Based on this interaction, we gave sufficient conditions for a test suite to be sound and exhaustive.

We gave two algorithms to construct global test cases: the fist one iteratively adds events to the test case while they do not introduce immediate conflict between inputs in the test case. Even if this algorithm takes as an input a linearization of the causality relation, we showed that the number of linearizations needed to cover the specification is exponential only in the number of direct conflicts between inputs which is usually considerably smaller than the number of all possible interleavings. However, this algorithm still generates redundant test cases: several linearizations can produce the same test case. In order to solve this, we proposed a second algorithm based on a SAT encoding of the test

cases.

The termination of both algorithms depends on the finiteness of the event structure that is used as an input (which is a prefix of the unfolding of the specification). We defined different criteria to construct finite prefixes of the unfolding and which can be seen as testing criteria. Defining testing criteria is necessary for covering certain behaviors of the system during the testing process since covering all is not usually possible (executing exhaustive test suites is not usually possible in practice since exhaustive test suites are usually infinite). Finally we showed that the test suite constructed by any of the defined criteria is sound and we proposed a definition of coverage to compare them.

# 6 A Distributed Testing Framework

Chapter 5 explains how to construct a global tester which controls and observes the whole system, however, global observation of the system cannot always be achieved and the testing process needs to be distributed. It is known that, in general, global traces cannot be reconstructed from local observations (see for example [BGMK07]), reducing the ability to distinguish different systems. There are three mainly investigated solutions to overcome this problem: *(i)* the conformance relation needs to be weakened considering partial observation; *(ii)* testers are allowed to communicate to coordinate the testing activity; *(iii)* stronger assumptions about the implementations are needed.

## 6.1 Conformance in Distributed Architectures

According to the three directions mentioned above, the following solutions have been proposed for testing global conformance in distributed testing architectures.

***(i)*** Hierons et al. [HMN08] argue that when the implementation is to be used in a context where the separate testers at the PCOs do not directly communicate with one another, the requirements placed on the implementation do not correspond to traditional conformance relations. In fact, testing the implementation using a method based on standard relations, such as **ioco**, may return an incorrect verdict. The authors of [HMN08] consider different scenarios, and a dedicated conformance relation for each of them. In the first scenario, there is a tester at each PCO which is independent from every other tester. In this scenario, it is sufficient that the local behavior observed at a PCO is consistent with some global behavior in the specification: this is captured by the **p-dioco** conformance relation. In the second scenario, a tester may receive information from other testers, and the local behaviors observed at different PCOs could be combined. Consequently, a stronger implementation relation called **dioco**,

is proposed. They show that **ioco** and **dioco** coincide when the system is composed of a single component, but that **dioco** is weaker than **ioco** when there are several components. Similar to this, Longuet [Lon12] studies different ways of globally and locally testing a distributed system specified with Message Sequence Charts [HT03], by defining global and local conformance relations. Moreover, conditions under which local testing is equivalent to global testing are established under trace semantics.

***(ii)*** Ulrich and König [UK99] propose a testing architecture where different testers are placed at each PCO. Those testers communicate via synchronization events and they assume that internal communication between processes is observable (grey-box testing). They also discuss several examples where these assumptions can be relaxed. Jard et al. [JJKV98] and Kang and [KK97] propose a method for constructing, given a global tester, a set of testers (one for each PCO) such that global conformance can be achieved by these testers. However, they also assume that testers can communicate with each other in order to coordinate the testing activity. In addition, Jard et al. consider the interaction between testers and the implementation as asynchronous.

***(iii)*** Bhateja and Mukund [BM08] propose an approach where they assume each component has a local clock and they append tags to the messages generated by the implementation. These enriched behaviors are then compared against a tagged version of the specification. Hierons et al. [HMN12] make the same assumption about the existence of local clocks. If the clocks agree exactly then the sequence of observations can be reconstructed. In practice the local clocks will not agree exactly, but some assumptions regarding how they can differ can be made. They explore several such assumptions and derive corresponding implementation relations.

This chapter proposes a formal framework for distributed testing of concurrent systems specified as network of automata, without relying on communications between testers. As networks of automata cannot model weak concurrency, we consider **co-ioco** as the global conformance relation to be tested. We show that some, but not all, situations leading to non global conformance can be detected by local testers without any further information about the other processes. Moreover we prove that when vector clocks [Fid88, Mat89] are used, the information held by each process suffices to reconstruct the global traces of an execution from the partial observations of it at each PCO, and that global conformance can thus be decided by distributed testers.

## 6.2 Modeling a Distributed System

A sound software engineering rule for building complex systems is to divide the whole system in smaller and simpler processes, each solving a specific task. This means that, in general, complex systems are actually collections of simpler

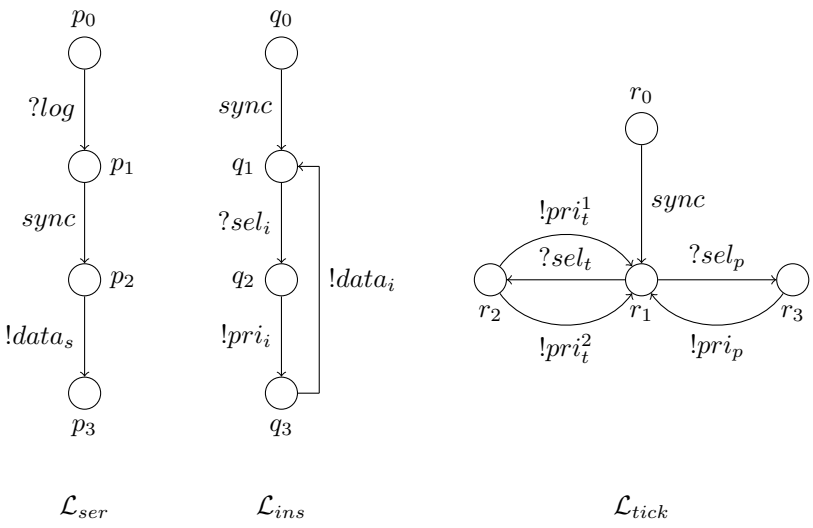



Figure 6.1: Network of automata of a travel agency.

processes running in parallel. We use automata (labeled transition systems with a finite set of states) to model local behaviors, while global behaviors are modeled by a collection or network of automata. We show that networks of automata are captured equivalently by Petri nets where explicit representation of concurrency avoids the state space explosion produced by interleavings.

We consider a distributed system composed of $n$ processes that communicate with each other over synchronization or communication actions and where the local model of a process is defined as a deterministic finite automaton. Several processes can communicate over the same communication action, but observable actions from different processes are disjoint[1], i.e. processes only share communication actions.

Example 6.1 (Network of Automata). *Figure 6.1 shows a network of automata with three processes that jointly represent the behavior of a travel agency and which synchronize over the action sync.*

**From Automata to Nets:** Given an automaton $\mathcal{L} = (Q, L, \Delta, q_0)$, its translation to a Petri net $\mathcal{N}_\mathcal{L} = (P, T, F, \lambda, M_0)$ is immediate: *(i)* places are the states of the automaton, i.e. $P = Q$; *(ii)* for every transition $(s_i, a, s_i') \in \Delta$ we add $t$ to $T$ and set ${}^\bullet t = \{s_i\}, t^\bullet = \{s_i'\}$ and $\lambda(t) = a$; *(iii)* the initial state is the only place marked initially, i.e. $M_0 = \{q_0\}$. By abuse of notation, we make no distinction between $\mathcal{L}$ and $\mathcal{N}_\mathcal{L}$.

The joint behavior of processes $\mathcal{L}_1, \ldots, \mathcal{L}_n$ is modeled by $\mathcal{N}_{\mathcal{L}_1} \times \cdots \times \mathcal{N}_{\mathcal{L}_n}$ where $\times$ represents the product of labeled nets [Fab06] and we only synchronize

[1] Action $a$ from process $\mathcal{L}_i$ is labeled by $a_i$ if necessary.

on communication transitions (which are invisible for the environment and thus labeled by $\tau$).

**Definition** 6.1 (Product of Nets). *Consider two nets $\mathcal{N}_1 = (P, T, F, \lambda, M_0)$ and $\mathcal{N}_2 = (P', T', F', \lambda', M_0')$ and $S$ the set of synchronization actions. The product $\mathcal{N}_1 \times \mathcal{N}_2$ and associated projections $\pi_i : \mathcal{N}_1 \times \mathcal{N}_2 \to \mathcal{N}_i$ are defined as follows*

- $\overline{P} = \{(p_1, \star) \mid p_1 \in P\} \cup \{(\star, p_2) \mid p_2 \in P'\}$*: disjoint union of places, $\pi_i(p_1, p_2) = p_i$ if $p_i \neq \star$ and it is undefined otherwise,*
- *the transition set $\overline{T}$ is given by*

$$\begin{array}{rcl} \overline{T} & = & \{(t_1, \star) \mid t_1 \in T, \lambda(t_1) \notin S\} \\ & \cup & \{(\star, t_2) \mid t_2 \in T', \lambda'(t_2) \notin S\} \\ & \cup & \{(t_1, t_2) \in T \times T' \mid \lambda(t_1) = \lambda'(t_2) \in S\} \end{array}$$

  *$\pi_i(t_1, t_2) = t_i$ if $t_i \neq \star$ and it is undefined otherwise,*
- *the flow $\overline{F}$ is defined by ${}^\bullet t = \pi_1^{-1}({}^\bullet \pi_1(t)) \cup \pi_2^{-1}({}^\bullet \pi_2(t))$ and analogously for $t^\bullet$, assuming ${}^\bullet \pi_i(t) = \pi_i(t)^\bullet = \emptyset$ if $\pi_i$ is undefined at $t$,*
- *$\overline{\lambda}$ is the unique labeling preserved by $\pi_i$,*
- $\overline{M}_0 = \pi_1^{-1}(M_0) \cup \pi_2^{-1}(M_0')$.

Product of nets prevents the state space explosion problem, as the number of places in the final net is linear w.r.t the number of components while product of automata produces an exponential number of states. This product allows to distinguish processes by means of its projections and when there are $n$ processes, it generates a distribution function $D : P \cup T \to \mathcal{P}(\{1, \ldots, n\})$ that relates each place and transition with its corresponding automata [vGGSU12]. In the case of communication actions, the distribution relates the synchronization transition with the automata that communicate over it. A net together with a distribution is called a distributed net. The unfolding of a distributed net is also distributed and it inherits its distribution, i.e. $\forall x \in B \cup E$ we have $D(x) = D(\varphi(x))$. By abuse of notation we make no distinction between the distribution in the original net and its unfolding.

Remark 5. *As different processes are deterministic and they only share communication actions, the net obtained by this product is deterministically labeled.*

Example 6.2 (Distributed Net). *Figure 6.2 shows the net obtained from the network of automata in Figure 6.1 and its distribution $D$ represented by colors. The transition corresponding to action ?log corresponds to the server process, i.e. $D(t_1) = \{ser\}$, while communication between the processes is converted into a single transition $t_2$ with $D(t_2) = \{ser, ins, tick\}$. This net adds places $r_0, q_0$ to the travel agency net $\mathcal{N}_{ag}$ of Figure 3.1 on page 28, however the behaviors of both nets are equivalent, i.e. they unfold into the same event structure $\mathcal{E}_{ag}$ of Figure 3.3 on page 32.*

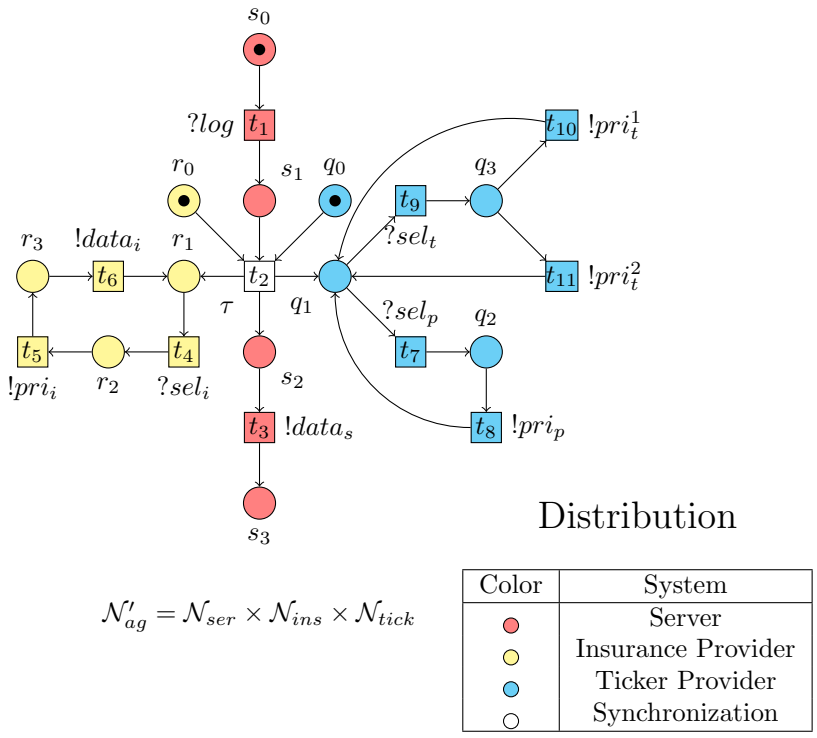



Figure 6.2: A distributed Petri net.

In a distributed net obtained from a network of automata, accepting an input or producing an output is decided locally, i.e. the structure of the net obtained by this product is such that inputs and outputs are controlled (enabled) by a single process.

**Proposition** 5. *For every process $\mathcal{N}_{\mathcal{L}}$ obtained from an automaton we have $\forall t \in T : |{}^\bullet t| = 1$. In the net obtained by the product between processes, this property is only violated by communication transitions. Therefore, any input or output event is enabled by exactly one place.*

*Proof.* Immediate from construction of $\mathcal{N}_{\mathcal{L}}$ and Definition 6.1. □

Remark 6. *The result of Proposition 5 is inherited by the unfolding of the net, i.e. every input or output event $e$ is such that $|{}^\bullet e| = 1$.*

The unfolding $\mathcal{U}$ of a distributed net together with its distribution $D$ allow to project the system over a single process by just considering the conditions and events in that process, i.e. conditions and events of projection $\mathcal{U}_d$ are $\{b \in B \mid d \in D(b)\}$ and $\{e \in E \mid d \in D(e)\}$ respectively. Similar to this, given an execution $\eta$ and an observation $\omega$ of the unfolding, we can project them over a given process $d$, the resulting projections are denoted by $\eta_d$ and $\omega_d$.

The notion of configuration represents the current global state of the system; in the case of a distributed system, the global state (configuration) is represented by the local state of each process.

Remark 7. *In a distributed system, every configuration $C$ of the unfolding generates a cut $C^\bullet = \{q_1, \dots q_n\}$ where each $q_d$ represents the current state of the $d^{th}$ process.*

Consider a given cut $C^\bullet = \{q_1, \dots, q_n\}$ and the configuration $C_d$ of process $d$ such that $C_d^\bullet = \{q_d\}$. An event which is not enabled in $C_d$ cannot be enabled in $C$, i.e. an event which is not locally possible is neither globally possible. The following result is central and will help proving that global conformance can be achieved by local testers.

**Proposition** 6. *Let $C$ (respectively $C_d$) be a configuration of a distributed system (respectively of the process $d$) with the corresponding cut $C^\bullet = \{q_1, \dots, q_n\}$ (respectively $C_d^\bullet = \{q_d\}$). Then:*

1. *if $?i_d \notin poss(C_d)$, then $?i_d \notin poss(C)$,*
2. *if $!o_d \notin out(C_d)$, then for all $!\omega \in out(C)$ we have $!\omega_d \neq !o_d$.*

*Proof.* If an input or output event is not enabled in configuration $C_d$, then there is no token in condition $q_d$ (see Remark 6). This absence prohibits such an event not only to be enabled in $C_d$, but also in $C$. □

Notice the distinction between possible inputs and produced outputs. Whenever the system reaches a configuration $C$ that enables input actions in every component, i.e. $?i_d \in \text{poss}(C_d)$ for all $d \in \{1, \dots, n\}$, from the global point of view, not only $?i_1$ **co** $\dots$ **co** $?i_n$ is possible for the system, but also every single input, i.e. $?i_d \in \text{poss}(C)$. The same is not true for produced outputs. Consider a system that enables output $!o_d$ in process $d$, leading to a quiescent configuration in that process, i.e. $!o_d \in \text{out}(C_d)$. If other components also enable outputs actions, $!o_d \notin \text{out}(C)$ as the global configuration after $!o_d$ is not quiescent.

Example 6.3 (Global System and its Projections). *Figure 6.3 shows the unfolding of net $\mathcal{N}'_{ag}$ and its projection $\mathcal{U}_{tick}$ over the ticket provider process. In the global system, each input and output is enabled by one condition, for example ${}^\bullet e_9 = \{q_1\}$, while communication events can have more than one condition in their preset as it is the case of $e_2$, i.e. ${}^\bullet e_2 = \{s_1, r_0, q_0\}$. Consider the reached configuration after logging in, synchronizing the processes and selecting a train, i.e. $C = \{\perp, e_1, e_2, e_9\}$, which generates the cut $C^\bullet = \{s_2, r_1, q_3\}$, i.e. after this trace, the ticket provider process is in state $q_3$. After this trace, it is not possible to select a plane in the ticket provider process, i.e. $?sel_p \notin poss(\{q_3\})$, and thus it is neither possible in the whole unfolding, i.e. $?sel_p \notin poss(C)$. The output $!pri_p$ is not produced in $\{q_3\}$ and thus it cannot be a projection of any output produced by the whole unfolding, i.e. $out(C) = \{!pri_t^1$ **co** $!data_s, !pri_t^2$ **co** $!data_s\}$ and $!pri_p \neq (!pri_t^1$ **co** $!data_s)_{tick}, !pri_p \neq (!pri_t^2$ **co** $!data_s)_{tick}$.*

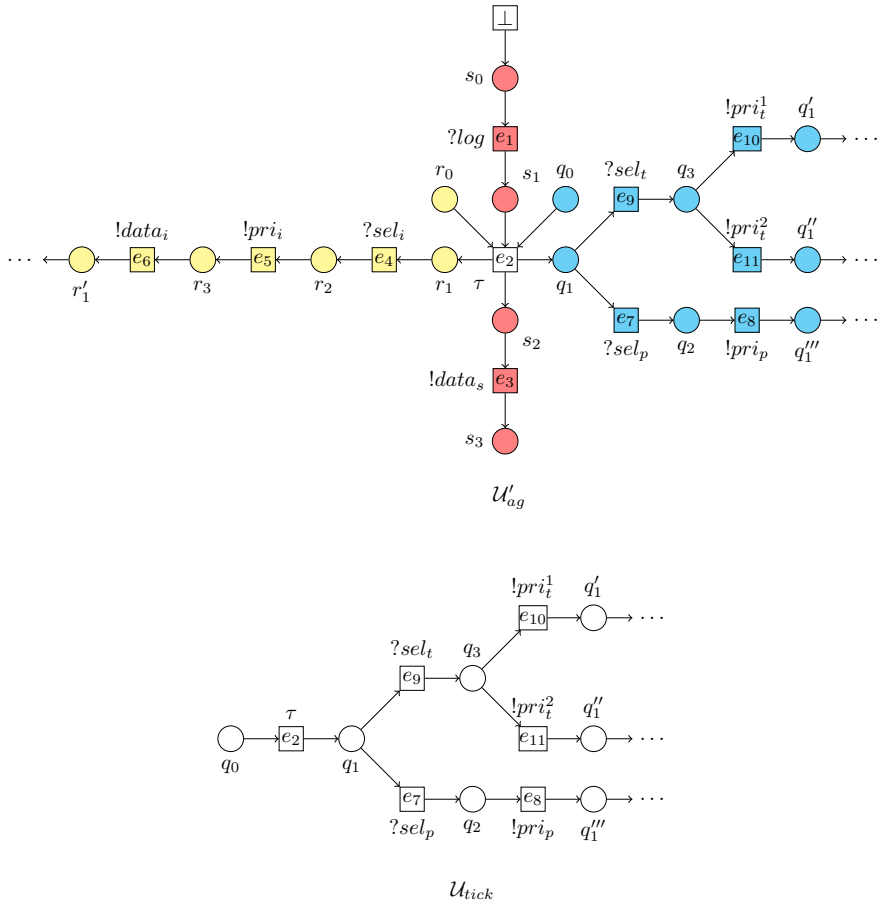



Figure 6.3: The unfolding of a distributed net and one of its projections.

## 6.3 Distributing Global Conformance

In this section we show how global conformance can be tested by placing a local tester at each PCO. We have seen that non conformance of the implementation is given by the absence of a given input or an unspecified output or quiescence in a configuration of the implementation. In a distributed system, a configuration defines the local state of each process as shown in Remark 7, thus, non conformance of a distributed system is due to one of the following reasons:

**(NC1)** An input which is possible in a state of a process in the specification is not possible in its corresponding state in the implementation,

**(NC2)** A state of a process in the implementation produces an output or a $\delta$ action while the corresponding state of the specification does not,

**(NC3)** The input (respectively output) actions that the configuration is ready

to accept (respectively produce) are the same in both implementation and specification, but they do not form the same partial order, i.e. concurrency is added or removed.

### 6.3.1 Detecting Non Conformance Locally

The observations collected at each PCO are rich enough to detect non conformance resulting from (NC1) and (NC2), i.e. those situations can be locally tested under **co-ioco** by transforming each component into a net as it is shown by the following result.

**Theorem** 12. *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and implementation of a distributed system, then $\mathcal{I}$* **co-ioco** *$\mathcal{S}$ implies that for every process $d \in \{1, \ldots, n\}$, $\mathcal{I}_d$* **co-ioco** *$\mathcal{S}_d$.*

*Proof.* Assume there exists $d \in \{1, \ldots, n\}$ for which $\neg(\mathcal{I}_d$ **co-ioco** $\mathcal{S}_d)$, then there exists $\sigma \in \text{traces}(\mathcal{S}_d)$ such that one of the following holds:

*(i)* There exists $?i \in \text{poss}(\mathcal{S}_d$ **after** $\sigma)$, but $?i \notin \text{poss}(\mathcal{I}_d$ **after** $\sigma)$. Consider the global trace of the specification $\omega = \langle ?i \rangle_{\mathcal{S}}$ which enables input $?i$ in $\mathcal{S}$, i.e. $?i \in \text{poss}(\mathcal{S}$ **after** $\omega)$. As $?i$ is not possible in $\mathcal{I}_d$, by Proposition 6 we have $?i \notin \text{poss}(\mathcal{I}$ **after** $\omega)$, and therefore $\neg(\mathcal{I}$ **co-ioco** $\mathcal{S})$.

*(ii)* There exists $!o \in \text{out}(\mathcal{I}_d$ **after** $\sigma)$ such that $!o \notin \text{out}(\mathcal{S}_d$ **after** $\sigma)$. Consider the global trace of the implementation $\omega = \langle !o \rangle_{\mathcal{I}}$ which enables $!o$ in $\mathcal{I}$, i.e. there exists $!\omega \in \text{out}(\mathcal{I}$ **after** $\omega)$ such that $!\omega_d = !o$. As $!o$ is not enabled in $\mathcal{S}_d$, by Proposition 6 we know that $!o$ cannot be enabled after $\omega$ in $\mathcal{S}$. Therefore, $!\omega \notin \text{out}(\mathcal{S}$ **after** $\omega)$ and $\neg(\mathcal{I}$ **co-ioco** $\mathcal{S})$.

*(iii)* $\delta \in \text{out}(\mathcal{I}_d$ **after** $\sigma)$, while $\delta \notin \text{out}(\mathcal{S}_d$ **after** $\sigma)$. Consider a trace $\omega$ which performs exactly the actions of $\sigma$ in process $d$ and takes every other process to a quiescent configuration (such trace always exists by Assumption 1); we have $\delta \in \text{out}(\mathcal{I}$ **after** $\omega)$. As the reached configuration in $\mathcal{S}_d$ after $\sigma$ is not quiescent and $\omega$ performs exactly the actions of $\sigma$ in that process, the specification enables some output in process $d$ which is also enabled at the global level and $\delta \notin \text{out}(\mathcal{S}$ **after** $\omega)$, therefore $\neg(\mathcal{I}$ **co-ioco** $\mathcal{S})$. □

Example 6.4 (Detecting Non Conformance Locally). *Figure 6.4 shows possible implementations of the ticket provider process given as automata and their corresponding unfoldings as event structures while the specifications of the ticket provider $\mathcal{U}_{tick}$ and whole system $\mathcal{U}'_{ag}$ are displayed in Figure 6.3. Implementation $\mathcal{I}_{tick}$ removes the possibility of selecting a train ticket, i.e. $?sel_t \notin poss(\mathcal{I}_{tick}$ **after** $\epsilon)$. The behavior of this ticket provider interacting with (correct) implementations of the other processes is shown by $\mathcal{I}'_6$. As selecting a train is a possible input in the specification of the ticket provider, we have that $\neg(\mathcal{I}_{tick}$* **co-ioco** *$\mathcal{U}_{tick})$ and by Theorem 12, we can conclude $\neg(\mathcal{I}'_6$* **co-ioco** *$\mathcal{U}'_{ag})$. Since the behavior of $\mathcal{U}'_{ag}$ is isomorphic to the event structure $\mathcal{I}_3 = \mathcal{E}_{ag}$ of Figure 3.3 and the observable behavior of $\mathcal{I}_6$ and $\mathcal{I}'_6$ coincide, we obtain the same result as in Example 3.12. The implementation of the ticket provider $\mathcal{I}'_{tick}$ produces an extra output after selecting a plane, i.e. $!pri^2_p \in out(\mathcal{I}'_{tick}$* **after** *$?sel_p)$,*

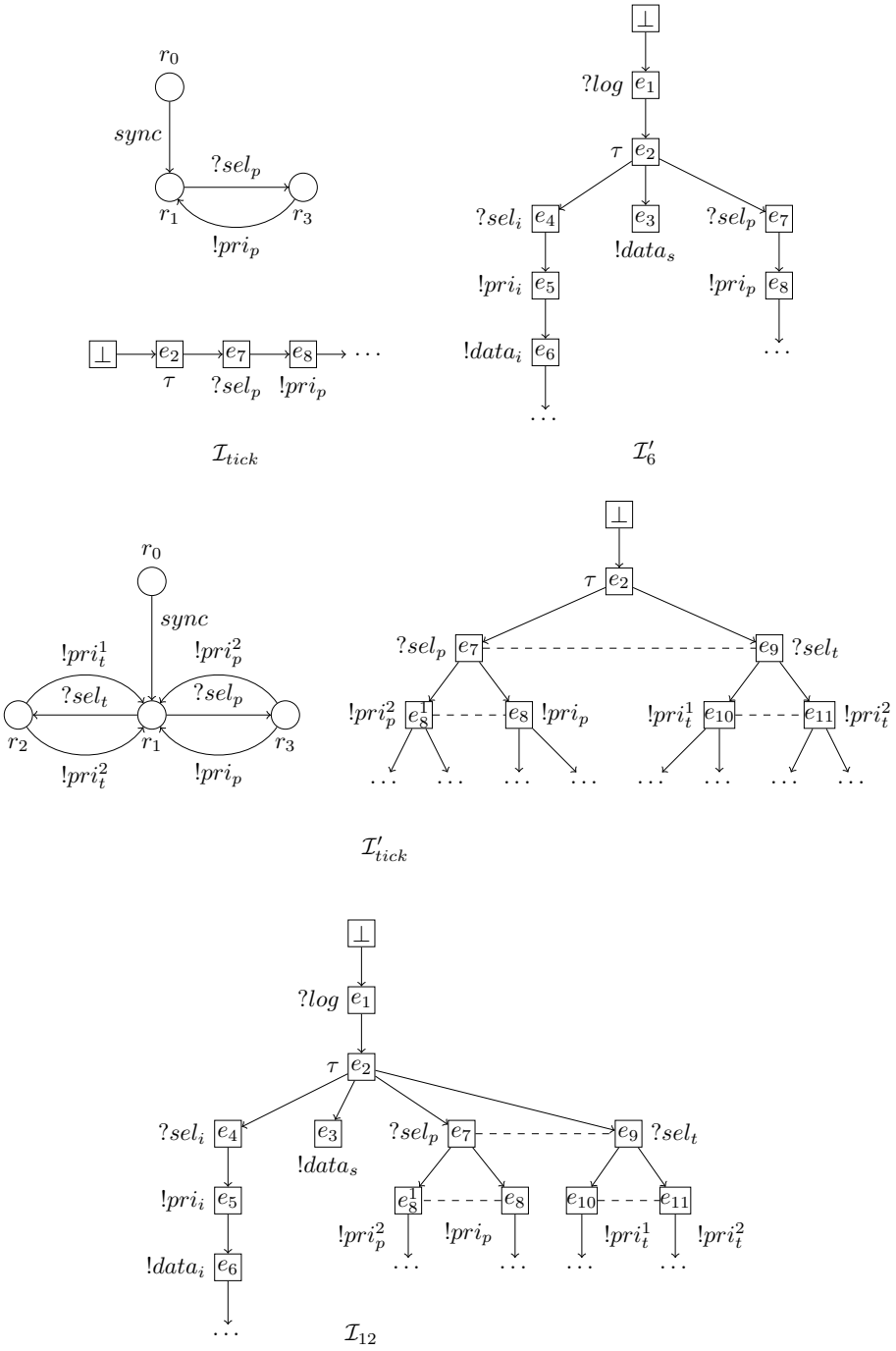
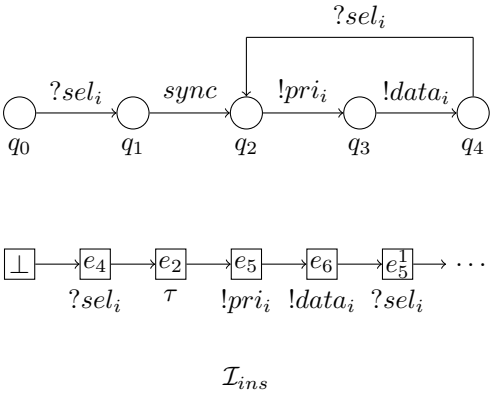



Figure 6.4: Detecting non conformance locally.

*while* $!pri_p^2 \notin out(\mathcal{U}_{tick}$ ***after*** $?sel_p)$ *and thus* $\neg(\mathcal{I}'_{tick}$ **co-ioco** $\mathcal{U}_{tick})$*. The joint behavior of this implementation with the rest of the processes is modeled by* $\mathcal{I}_{12}$ *and by Theorem 12 we have* $\neg(\mathcal{I}_{12}$ **co-ioco** $\mathcal{U}'_{ag})$*. It is easy to see that this result is correct as one of the outputs produced by* $\mathcal{I}_{12}$ *after logging in and selecting a plane ticket is not specified, i.e.* $out(\mathcal{I}_{12}$ ***after*** $?log{\cdot}?sel_p) = \{!data_s$ **co** $!pri_p, !data_s$ **co** $!pri_p^2\}$ *while* $!data_s$ **co** $!pri_p^2 \notin out(\mathcal{U}'_{ag}$ ***after*** $?log{\cdot}?sel_p)$*.*



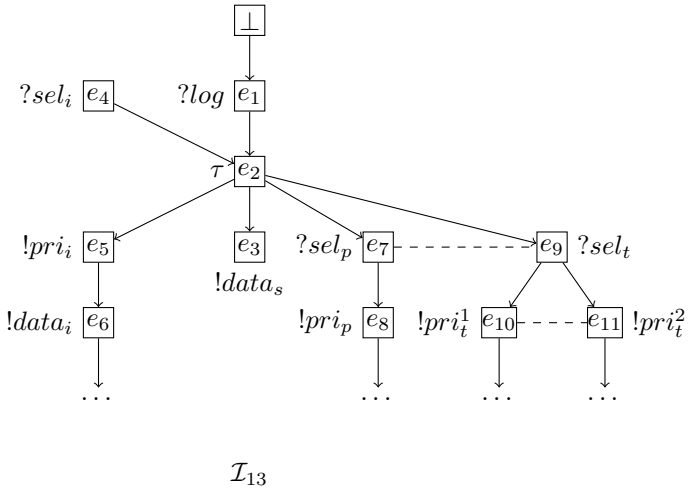




Figure 6.5: Extra causalities between processes

Example 6.5 (Extra Causalities Between Processes). *Theorem 12 gives necessary conditions for testing conformance locally, however even if every process is locally conformant, there can be extra dependencies between them that cannot be detected. Consider the implementation of the insurance provider process* $\mathcal{I}_{ins}$ *in Figure 6.5 which allows to select the insurance before synchronizing with the*

*rest of the processes. This implementation is correct w.r.t the specification* $\mathcal{L}_{ins}$ *of Figure 6.1 as the observable behaviors of both systems are equivalent (it is possible to select the insurance and after it, both outputs are produced). Implementation* $\mathcal{I}_{13}$ *shows the joint behavior of* $\mathcal{I}_{ins}$ *with the rest of the processes. This implementation is not conformant w.r.t the specification* $\mathcal{U}'_{ag}$ *as the input* $?sel_t$ *is not possible after just logging in, i.e.* $?sel_t \in poss(\mathcal{U}'_{ag}$ ***after*** $?log)$*, but* $?sel_t \notin poss(\mathcal{I}_{13}$ ***after*** $?log)$ *and thus* $\neg(\mathcal{I}_{13}$ **co-ioco** $\mathcal{U}'_{ag})$*.*

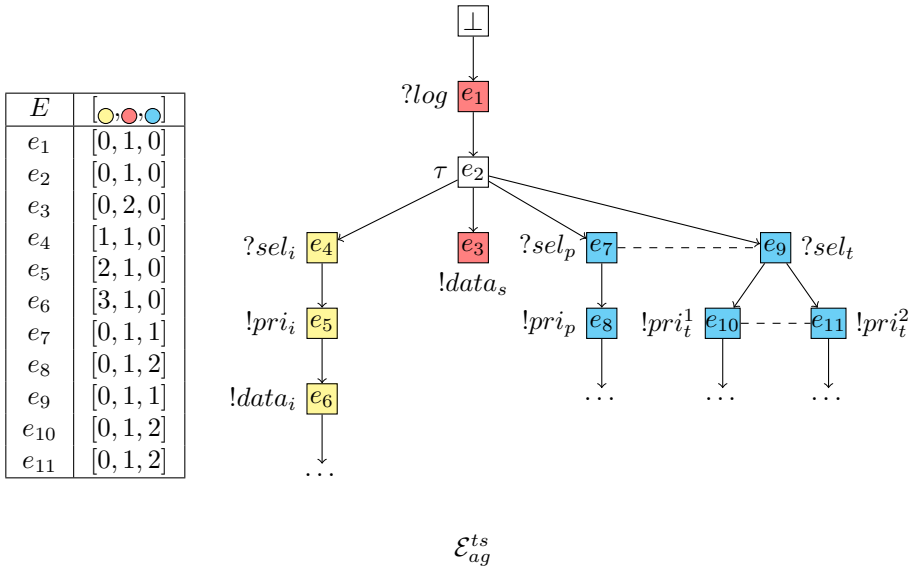
### 6.3.2 Adding Time Stamps

The example above shows that global conformance cannot always be achieved by testing each process locally. This is exactly what happens in situation (NC3). However, as processes of the implementation need to synchronize during communication, we propose to use such synchronization to interchange some information that allows the testers to recompute the partial order between actions in different processes using vector clocks [Fid88, Mat89].

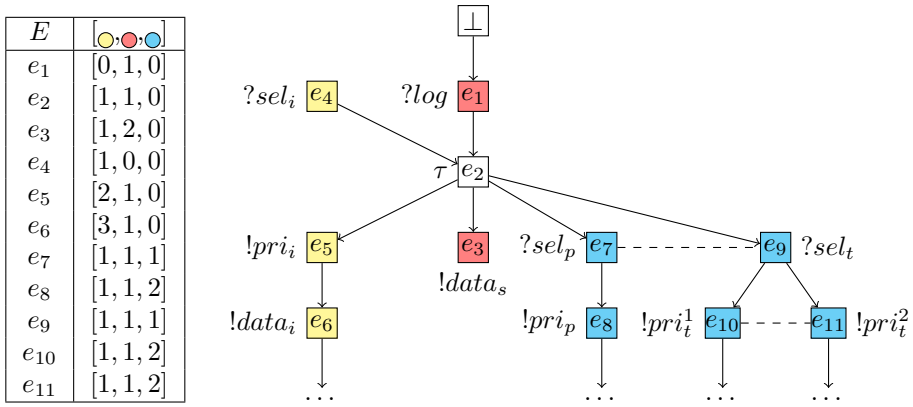
We assume each process counts the number of interactions between itself and the environment, and it stores it in a local table with information about every process where information about other processes may not be updated. Each process has a table of the form $[ts_1, \ldots, ts_n]$ and whenever two processes communicate via synchronization, their local tables are updated. In addition, we assume that the information of the table of each process is observable at each PCO by the tester which can also observe the time information of the actions. This assumptions imply that the tester knows exactly the values of the tables before proposing an input and the local time in which an output was produced together with the dependences coming from other processes.

We add the information about the tables to the model, i.e. events of the unfolding are tuples representing both the actions and the current value of the table. The unfolding algorithm allows to compute such tables easily: when event $e$ occurs, the $d^{th}$ entry in its table is equal to the number of input and outputs events from process $d$ in the past of $e$, i.e. $ts_d(e) = \mid [e] \cap (E_d^{\mathcal{I}n} \uplus E_d^{\mathcal{O}ut}) \mid$. The unfolding method (Algorithm 2) can be modified to consider time stamps resulting in Algorithm 5. The behavior of system $\mathcal{E}$ where time stamps are considered is denoted by $\mathcal{E}^{ts}$.

Example 6.6 (Time Stamped Unfoldings). *Consider the time stamped unfolding* $\mathcal{E}^{ts}_{ag}$ *of Figure 6.6 where the tables of the form* $[ins, ser, tick]$ *represent respectively the interactions with the environments of the insurance provider, the server and the ticket provider. The* $?log$ *action (event* $e_1$*) is the first interaction of the server with the environment and this can be seen in the second component of its table. As there is no synchronization before this action, the table does not contain any information about the other processes, i.e. their values are* 0*. The values of the table of* $e_4$ *contain the following information:* $?sel_i$ *is the first interaction between the insurance provider and the environment and when it occurs, we are sure that* $?log$ *already occurred as one action from the server should precede it. However, as* $!data_s$ *is the second interaction of the server with the*




$\mathcal{E}_{ag}^{ts}$




$\mathcal{I}_{13}^{ts}$

Figure 6.6: Time stamped unfoldings.

**Algorithm 5** Time stamped unfolding algorithm

**Require:** a Petri net $\mathcal{N} = (P, T, F, \lambda, M_0)$, where $M_0 = \{p_1, \ldots, p_k\}$ and a distribution $D : T \cup P \to \{1, \ldots, n\}$

**Ensure:** the time stamped unfolding of $\mathcal{N}$

1: $\mathcal{U} := \{(\bot, \emptyset), (p_1, \bot), \ldots, (p_k, \bot)\}$
2: $pe := PE(\mathcal{U})$
3: **while** $pe \neq \emptyset$ **do**
4: chose an event $e = (t, C)$ in $pe$
5: **for** $d \in \{1, \ldots, n\}$ **do**
6: $ts_d(e) := |\{(t', C') \in \mathcal{U} \mid D(t') = d \wedge \lambda(t') \neq \tau\}| + 1$
7: add to $\mathcal{U}$ the event $e \times ts_1(e) \times \cdots \times ts_n(e)$ and a condition $(p, e)$ for every place $p$ in $t^\bullet$
8: $pe := PE(\mathcal{U})$
9: **return** $\mathcal{U}$

*environment and we only know that one interaction precedes $?sel_i$, we cannot get any information about the relation between these actions which is consistent with their independence (concurrency). Synchronization events do not increase by their own the values in the tables, but they update the values. Consider event $e_2$ in $\mathcal{I}_{13}^{ts}$ and its table $[1, 1, 0]$. The information that one action occurred in the insurance provider and another in the server is propagated to the tables of every event that causality depends on $e_2$, for example $e_3, e_5, e_7, e_9$.*

Time stamps allow to reconstruct the global trace of an execution of the system from the local traces of this execution observed at PCOs. Causalities coming from the same process are observed locally, while causalities between actions at different processes are recomputed using the time stamped information.

Example 6.7 (From Sequences to Partial Orders Using Time Stamps). *Consider the time stamped local traces of implementation $\mathcal{I}_{13}^{ts}$ on Figure 6.7 (above). From the event $(!data_s, 1, 2, 0)$, we know that at least one event from the insurance provider process precedes $!data_s$ and as $?sel_i$ is the first action on this process, we can add the causality $(?sel_i, 1, 0, 0) \leq (!data_s, 1, 2, 0)$ as it is shown in the partial order $\omega$. The other three causalities between actions belonging to different processes can be added following the same reasoning. It can be observed that the resulting partial order $\omega$ is a partial order observation of the global system, i.e. $\omega \in traces(\mathcal{I}_{13}^{ts})$.*

Given two time stamped LPOs $\omega_i = (E_i, \leq_i, \lambda_1)$ and $\omega_j = (E_j, \leq_j, \lambda_2)$, their joint causality is given by the LPO $\omega_i + \omega_j = (E_i \uplus E_j, \leq_{ij}, \lambda_1 \uplus \lambda_2)$ where for each pair of events $e_1 = (a, t_1, \ldots, t_n) \in E_i$ and $e_2 \in E_i \uplus E_j$, we have

$$e_2 \leq_{ij} e_1 \Leftrightarrow e_2 \leq_i e_1 \vee |[e_2]_j| \leq t_j$$

In other words, $e_2$ globally precedes $e_1$ either if they belong to the same process and $e_2$ locally precedes $e_1$ or if $e_2$ is the $t_j^{th}$ event in process $j$ and $e_1$ is preceded by at least $t_j$ events in component $j$ according to time stamps.

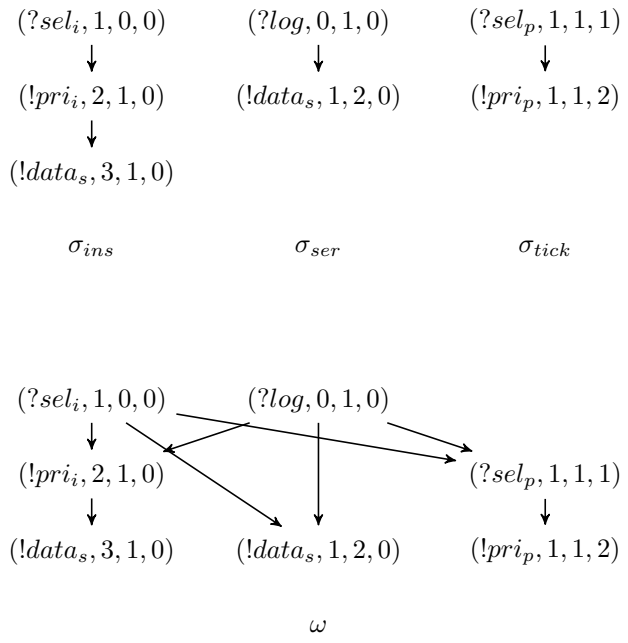



Figure 6.7: From sequences to partial orders using time stamps.

When communication between processes is asynchronous, a configuration $C$ is called consistent if for every sending message in $C$, its corresponding receive message is also in $C$. Mattern [Mat89] shows that consistent configurations have unambiguous time stamps; hence global causality can be reconstructed from local observations in a unique way. Under synchronous communication, the send and receive actions are represented by the same event, and therefore every configuration is consistent. This allows to prove the following results.

**Proposition** 7. *When the communication between processes is synchronous, the partial order obtained by $+$ is unique.*

*Proof.* Since every configuration is consistent, the result is immediate following the result of [Mat89]. □

Non conformance coming from (NC3) can be detected by testing the time stamped system in a distributed way.

**Theorem** 13. *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and implementation of a distributed system, then $\forall d \in \{1, \ldots, n\}$ : $\mathcal{I}_d^{ts}$ **co-ioco** $\mathcal{S}_d^{ts}$ implies $\mathcal{I}$ **co-ioco** $\mathcal{S}$.*

*Proof.* Assume $\mathcal{I}_d^{ts}$ **co-ioco** $\mathcal{S}_d^{ts}$ for every $d \in \{1, \ldots n\}$. Let $\omega \in \text{traces}(\mathcal{S})$ and consider the following situations:

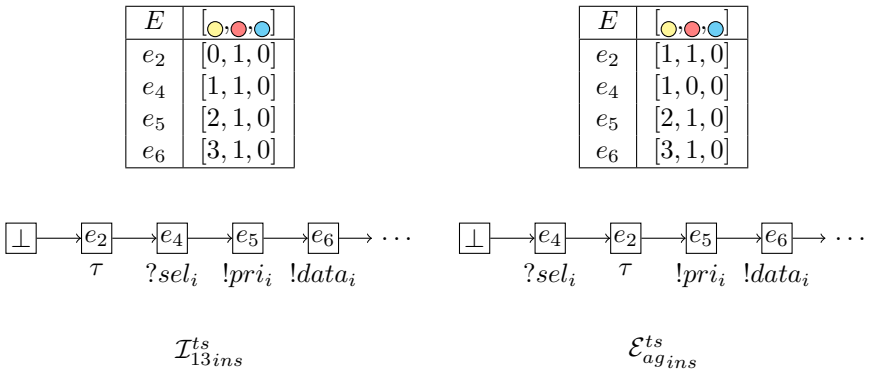
*(i)* If $?\omega \in \text{poss}(\mathcal{S}$ **after** $\omega)$, then for every $d$ there exists a time stamped input $(?i_d, t_1, \ldots, t_n) \in \text{poss}(\mathcal{S}_d^{ts}$ **after** $\omega_d)$ such that $?\omega_d = ?i_d$ and $?\omega =$

$?i_1 + \cdots + ?i_n$. As for all $d$, we have $\mathcal{I}_d^{ts}$ **co-ioco** $\mathcal{S}_d^{ts}$, then $(?i_d, t_1, \ldots, t_n) \in \text{poss}(\mathcal{I}_d^{ts}$ **after** $\omega_d)$. By Proposition 7, $?\omega \in \text{poss}(\mathcal{I}$ **after** $\omega)$.

*(ii)* If $!\omega \in \text{out}(\mathcal{I}$ **after** $\omega)$, then for every $d$ there exists a time stamped output $(!o_d, t_1, \ldots, t_n) \in \text{out}(\mathcal{I}_d^{ts}$ **after** $\omega_d)$ such that $!\omega_d = !o_d$ and $!\omega = !o_1 + \cdots + !o_n$. As every process of the implementation conforms to its specification, we have $(!o_d, t_1, \ldots, t_n) \in \text{out}(\mathcal{S}_d^{ts}$ **after** $\omega_d)$. By Proposition 7, we have $!\omega \in \text{out}(\mathcal{S}$ **after** $\omega)$.

*(iii)* If $\delta \in \text{out}(\mathcal{I}$ **after** $\omega)$, as the reached configuration by the global implementation after $\omega$ is quiescent, so there are the reached configurations of every local processes, i.e. $\delta \in \text{out}(\mathcal{I}_d^{ts}$ **after** $\omega_d)$. Since $\mathcal{I}_d^{ts}$ **ioco** $\mathcal{S}_d^{ts}$ for each $d$, we have $\delta \in \text{out}(\mathcal{S}_d^{ts}$ **after** $\omega_d)$. This implies that the local configurations of the specification do not enable any output; by Remark 6, outputs can only be enabled locally, thus there is no output enabled in the global configuration and $\delta \in \text{out}(\mathcal{S}$ **after** $\omega)$.

These three cases allow us to conclude that $\mathcal{I}$ **ioco** $\mathcal{S}$. □




$\mathcal{I}_{13_{ins}}^{ts}$ $\mathcal{E}_{ag_{ins}}^{ts}$

Figure 6.8: Detecting extra causality locally with time stamps.

Example 6.8 (Detecting Non Conformance Locally with Time Stamps). *Consider the time stamped unfoldings of $\mathcal{I}_{ins}$ and the specification of the insurance process of Figure 6.8. Both systems are the projections of $\mathcal{I}_{13}^{ts}$ and $\mathcal{E}_{ag}^{ts}$ over the insurance process. We can see that even if selecting an insurance is possible in both the implementation and the specification, their time stamps do not coincide, i.e. $(?sel_i, 1, 0, 0) \in poss(\mathcal{E}_{ag_{ins}}^{ts}$ **after** $\epsilon)$, but $(?sel_i, 1, 0, 0) \notin poss(\mathcal{I}_{13_{ins}}^{ts}$ **after** $\epsilon) = \{(?sel_i, 1, 1, 0)\}$, and then we have $\neg(\mathcal{I}_{13_{ins}}^{ts}$ **co-ioco** $\mathcal{E}_{ag_{ins}}^{ts})$. By Theorem 13, we can conclude $\neg(\mathcal{I}_{13}$ **co-ioco** $\mathcal{E}_{ag})$ which is consistent with the result of Example 6.5 as $\mathcal{E}_{ag}$ is isomorphic to $\mathcal{U}'_{ag}$.*

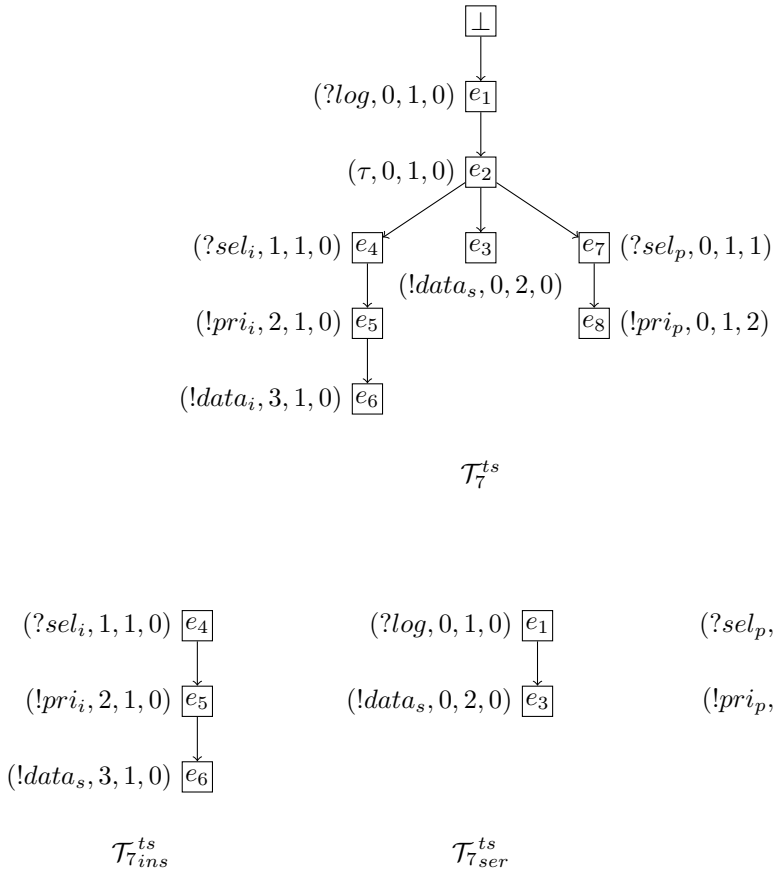



Figure 6.9: From global test cases to distributed ones.

## 6.4 From Global Test Cases to Distributed Ones

We have shown in the last section that global conformance can be tested in a distributed way, i.e. by testers located at each PCO. However testers need to consider time stamp information which cannot be computed locally. The test case generation method that we proposed in Chapter 5 can easily be adapted to add time stamps and the global test case obtained can be projected to each process to obtain a set of distributed test cases.

Example 6.9 (From Global Test Cases to Distributed Ones). *Consider the test case $\mathcal{T}_7^{ts}$ of Figure 6.9 obtained by considering, for example, a "all path of length 3" criterion, the time stamped unfolding and the SAT encoding method to the specification $\mathcal{N}'_{ag}$. This global test case can be projected[2] into the local test cases $\mathcal{T}_{7_{ins}}^{ts}, \mathcal{T}_{7_{ser}}^{ts}$ and $\mathcal{T}_{7_{tick}}^{ts}$ which are executed in parallel at each PCO of $\mathcal{I}_{13}^{ts}$. Consider the test case $\mathcal{T}_{7_{ser}}^{ts}$ placed at the PCO of the server. As testers*

[2]Internal actions can be abstracted since they are unobservable for the tester.

*do not communicate, it can be the case that when* $?log$ *is sent in this PCO,* $?sel_i$ *was already sent by the other tester and therefore the output* $!data_s$ *is produced. However the time stamp of this output does not coincide with the time stamp expected by the tester, i.e.* $(!data_s, 1, 2, 0) \in out(\mathcal{I}^{ts}_{13_{ser}}$ ***after*** $?log)$*, but* $(!data_s, 1, 2, 0) \notin out(\mathcal{T}^{ts}_{7_{ser}}$ ***after*** $?log) = \{(!data_s, 0, 2, 0)\}$*, therefore the test execution blocks and the execution fails, detecting the non conformance of implementation* $\mathcal{I}_{13}$*.*

**Controllability and observability problems:** We already explained in the introduction that distribution of the testing architecture can lead to situations where the tester does not know when to apply an input or if a produced output has extra or missing dependencies from other processes. Clearly, these problems do not exists in a global test case as a global control is assumed. The problem is also solved when we use vector clocks in the construction of test cases.

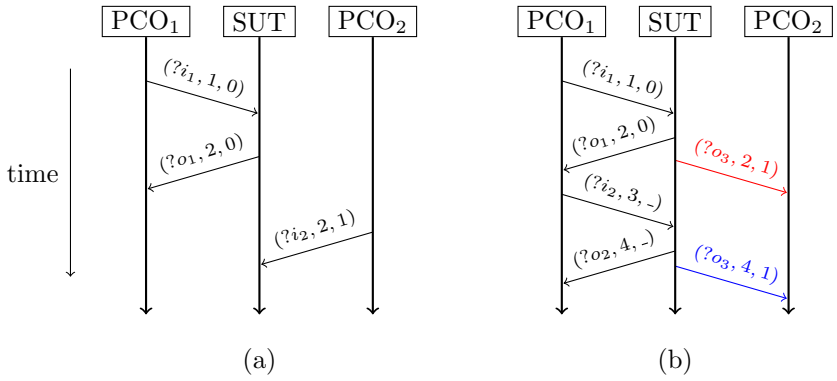



Figure 6.10: Controllability and observability problems in distributed architectures.

Example 6.10 (Controllability and Observability Problems). *Consider the situation (a) in Figure 6.10. When time stamps are added, the global test case has an unique trace* $(?i_1, 1, 0) \cdot (!o_1, 2, 0) \cdot (?i_2, 2, 1)$ *which can be projected into the local test case* $(?i_2, 2, 1)$ *for the second PCO. As we assume that the implementation has clocks and information about the clock of each process is updated during syntonization, the tester at* $PCO_2$ *needs to wait until its information is updated to have the certainty that two observable actions have already occurred at* $PCO_1$*. The situation of (b) can also be detected since the outputs* $(!o_3, 2, 1)$ *and* $(!o_3, 4, 1)$ *can easily be differentiated.*

## 6.5 Conclusion

Distributed systems can naturally be modeled by networks of automata where each automaton represents the behavior of a single process. We showed how to

obtain a Petri net from such network when communication between processes is synchronous. This net represents the global behavior of the system and can be tested using the theory presented in the previous chapters of this thesis.

We showed that some of the reasons that lead to non conformance can be tested locally by transforming each process into a concurrency-free net and testing it w.r.t **co-ioco**. However, certain situations that lead to non conformance remain undetected, this happens when the error comes from some extra or missing dependence between processes. Since these dependences are not observable from the environment, we need extra machinery to detect them. We showed that under the assumption that each process of the implementation has a local clock that counts the number of interactions between itself and the environment, vector clocks can be used to reconstruct global traces from local ones and thus global conformance can be achieved by local testers.

Local testers enriched with information about time stamps do not only allow to detect all the causes of non conformance, but they also solve the controllability and observability problems generated by the distribution of the testing architecture. However, in order to compute the time stamp information, global test cases need to be constructed before projecting them onto the local test cases that are actually executed in the implementation.

# 7 Conclusions and Perspectives

## 7.1 Summary

Even if models that handle true concurrency have been used in the past for test case generation to avoid the state space explosion problem, the independence of actions never played a central role in testing concurrent systems. This thesis introduces two conformance relations (**co-ioco** and **wsc-ioco**) where independence of actions is preserved in any correct implementation. We present the well-known **ioco** relation based on a behavior model and extend it with the test of refusals (input-enabledness of the implementation is not assumed) to compare it with the other relations presented in this thesis.

In Chapter 3 we extended the observation notions used by the **ioco** conformance relation to true concurrency models like Petri nets and their unfoldings. These notions include traces, refusals, produced outputs and quiescence allowing us to present the **co-ioco** conformance relation that preserves independence between actions. We showed that **co-ioco** is a generalization of **ioco** in the sense that when there is no concurrency and the implementation is input-enabled both relations coincide.

Chapter 4 presents a new semantics for concurrent systems that can be seen as an intermediate point between interleaving and partial order semantics. This semantics generalized the other two: whenever there is no weak concurrency, they boil down to partial order semantics, but if there is no strong concurrency, the new semantics is equivalent to interleaving semantics. The last remarks allow to relate the three conformance relations presented in this thesis.

We developed in Chapter 5 a complete testing framework for these conformance relations, this includes the definition of global test cases (which reduces both the size and the number of test cases needed to test the implementation) and their execution on the system. We present a first algorithm that constructs test cases by iteratively adding events. Even if this method requires to use linearizations of the causality relation, which would reduce the gain of using

partial order semantics, we show that the complexity of the method is bounded by a factor that is exponential only in the immediate conflicts between inputs of the system. However, this approach still generates some redundant test cases; to overcome this, we propose a SAT encoding for the generation of test cases. Both test generation algorithms are based on the unfolding of the system; as such unfolding is usually infinite, this generates an infinite test suite. We propose to select a finite and sound set of test cases based on different testing criteria. These criteria are based on the structure of the specification, i.e. path of certain length and cycles. We show that each testing criterion corresponds to a different cut-off method and we give a parametric unfolding algorithm based on such criteria.

Chapter 6 deals with distributed testing architectures and shows that some situations leading to non conformance can be detected locally (this is the case when the error belongs locally to a process), however errors coming from the interaction of the processes cannot be detected locally. We show that under the assumption that each process has a local clock, we can reconstruct global traces from local ones, using vector clocks, and therefore global conformance can be achieved locally.

## 7.2 Future Research

Although this thesis tackles several issues of model-based conformance testing for concurrent system, there are plenty of challenges that need further investigation.

Section 5.1.2 formalizes the cases where the interaction between the implementation and a global test case blocks, however it is defined in terms of possible inputs and produced outputs while when systems are modeled by LTS, test execution is formalized by the product of LTSs. Since product of nets does not preserve concurrency, such product cannot be used for modeling the interaction of a test cases with the implementation. Suppose two actions are concurrent in the test case and ordered in the implementation, the product allows to execute them, but ordered instead of concurrently as was expected by the test case. If a process has information about when actions in another process are ready to fire, it knows which actions are concurrent to its own actions. Adding read arcs to the product of nets and using step semantics allow to model test execution while preserving concurrency.

As it is mentioned in the introduction, some of the testing assumptions are essential: conformance is not possible without them, while others are just practical. The latter is the case of the assumption about conflict between input and outputs we made in Section 5.2.2. We make such an assumption since it allows us to use Theorem 6 and then prove soundness of the test suite. However the theorem gives sufficient (but not necessary) conditions for a test suite to be sound. Relaxing such technical assumption is part of our future work. Another testing assumption made in this thesis is that the implementation is a black box. It has been shown [UK99] that when the communication between

processes is observable in the implementation, distributed testers at each PCO work correctly and bring up the expected result. Considering observable communication can help us to solve the testing problem in distributed architectures without using vector clocks.

The test selection problem was solved in this thesis following different criteria that allow a finite representation of the state space of the specification. Other approaches (like [JJ05]) use test purposes for solving such a problem. Event structures seem as a promising formalism to define test purposes in the concurrent setting. This thesis also solves the distribution of the testing architecture by making stronger assumptions on the implementation in order to reconstruct global traces from local ones. Another possible solution is to define new notions of conformance for systems specified with Petri nets or network of automata and that take distribution into account; this new relations need to be compared with the relations presented in this thesis. The coordination of the testing activity can also solve the distribution problem: a set of local testers can be seen as a global tester where actions between testers are concurrent. However, further information need to be added to this global test in order to coordinate the testing activity by imposing the preservation of independence between some actions. Petri nets with read arcs [Vog02] seem to be an interesting model for modeling such coordination since their step semantics allows to force concurrency.

As it was the case with **ioco**, we hope that the theory of this thesis opens the door to new research in testing enriched concurrent systems using time [PZ13], probability [BK96] or read arcs [Vog02, Rod13].

From the practical point of view, even if the algorithms presented in this thesis have been implemented and the prototype allows us to do some experiments (see Appendix B), it does still not consider the results of Chapter 6, i.e. the distribution of the architecture and projection of global test cases into local test cases. Also, the test cases generated by the tool are abstract (they are expressed by event structures), while other tools such as JTorX [Bel10] implement the **ioco** theory and are capable to transform abstract test cases in inputs to a real implementation and automatically execute them. The concretization of test cases and automatic execution in real software implementations is the next step of our tool.

# A

# Other Conformance Relations

This thesis presented extensions of the **ioco** conformance relation for reactive systems, where a distinction is made between actions proposed by the environment and the responses produced by the system. However, the implementation relations that were first presented in the literature did not consider such distinction; communication between the system and its environment was modeled by parallel composition where the interaction was symmetric. Whenever a system wished to interact with its environment, it proposed some actions on which it was prepared to interact. The environment also proposed some actions, and then they interacted on one of those actions that both proposed. The role of both communication processes is the same and symmetric, and all observable actions were treated in the same way.

Implementation relations for LTS based on symmetric interactions include among others observation equivalence [Mil80], strong bisimulation and weak bisimulation equivalence [Par81, Mil89, Abr87], failure equivalence and preorder [Hoa85], testing equivalence and preorder [DH84], failure trace equivalence and preorder [BW90]. Depending on the chosen semantics for concurrency, testing equivalences [ADF86], and both trace and bisimulation equivalences [vG89] have been proposed for ES.

One of the easiest way to reason about the behavior of a system is in terms of its traces. The definition of traces does not depend on the way the system and its environment communicate (symmetrically or anti-symmetrically) and it can be used for systems where there is no distinction between input and output actions. However, the notion of trace still depends on the chosen semantics for concurrency.

## A.1 Sequential Executions

Chapter 3 defines partial order traces for an event structure. However, event structures also admit interleaving semantics where one goes from one configu-

ration to another by performing just an action at a time. Sequential executions are captured by the following definition.

**Definition** A.1 (Sequential Execution of an ES). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ with $C, C' \in \mathcal{C}(\mathcal{E})$ and $\mu, \mu_i \in (L \cup \{\tau\})$, we define*

$$\begin{array}{lcl} C \xrightarrow{\mu}_s C' & \triangleq & \exists e \in E : C' = C \uplus \{e\} \text{ and } \lambda(e) = \mu \\ C \xrightarrow{\mu_1 \dots \mu_k}_s C' & \triangleq & \exists C_0, \dots, C_k : C = C_0 \xrightarrow{\mu_1}_s C_1 \xrightarrow{\mu_2}_s \dots \xrightarrow{\mu_k}_s C_k = C' \\ C \xrightarrow{\mu_1 \dots \mu_k}_s & \triangleq & \exists C' : C \xrightarrow{\mu_1 \dots \mu_k}_s C' \end{array}$$

*We say that $\mu_1 \dots \mu_k$ is a sequential execution of $C$ if $C \xrightarrow{\mu_1 \dots \mu_k}$.*

The notion of sequential observation, traces and reached configurations are straightforward.

**Definition** A.2 (Observations, Traces and Reached Configurations). *Let $\mathcal{E} = (E, \leq, \#, \lambda) \in \mathcal{ES}(L)$ with $C, C' \in \mathcal{C}(\mathcal{E}), a, a_i \in L$ and $\sigma \in L^*$, we define*

$$\begin{array}{lcl} C \stackrel{\epsilon}{\Longrightarrow}_s C' & \triangleq & C = C' \text{ or } C \xrightarrow{\tau \dots \tau}_s C' \\ C \stackrel{a}{\Longrightarrow}_s C' & \triangleq & \exists C_1, C_2 : C \stackrel{\epsilon}{\Longrightarrow}_s C_1 \xrightarrow{a}_s C_2 \stackrel{\epsilon}{\Longrightarrow}_s C' \\ C \stackrel{a_1 \dots a_k}{\Longrightarrow}_s C' & \triangleq & \exists C_0, \dots, C_k : C = C_0 \stackrel{a_1}{\Longrightarrow}_s C_1 \stackrel{a_2}{\Longrightarrow}_s \dots \stackrel{a_k}{\Longrightarrow}_s C_k = C' \\ C \stackrel{\sigma}{\Longrightarrow}_s & \triangleq & \exists C' : C \stackrel{\sigma}{\Longrightarrow}_s C' \\ traces_s(\mathcal{E}) & \triangleq & \{\sigma \in L^* \mid \perp \stackrel{\sigma}{\Longrightarrow}_s\} \\ C \ \textbf{\textit{after}}_s \ \sigma & \triangleq & \{C' \mid C \stackrel{\sigma}{\Longrightarrow}_s C'\} \end{array}$$

We now define conformance relations for both sequential and partial order semantics of event structures when interaction is symmetric. For the complete definitions of sequences, traces and reached configurations, see Section 3.2.3 and Section 3.3.1.

## A.2 Trace Preorder

The first relation we present, called trace preorder, is based on the inclusion of the observable executions of the system under test in those allowed by the specification. The intuition is that an implementation $\mathcal{I}$ should not exhibit an unspecified behavior, i.e. not present in $\mathcal{S}$.

**Definition** A.3 (Trace Preorder). *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and implementation of the system, then*

$$\begin{array}{lcl} \mathcal{I} \leq^s_{tr} \mathcal{S} & \Leftrightarrow & traces_s(\mathcal{I}) \subseteq traces_s(\mathcal{S}) \\ \mathcal{I} \leq_{tr} \mathcal{S} & \Leftrightarrow & traces(\mathcal{I}) \subseteq traces(\mathcal{S}) \end{array}$$

Example A.1 (Trace Preorder). *Consider Figure A.1. Under partial order semantics, the traces of $\mathcal{E}_1$ and $\mathcal{E}_2$ coincide, then $\mathcal{E}_1 \leq_{tr} \mathcal{E}_2$ and $\mathcal{E}_2 \leq_{tr} \mathcal{E}_1$. The traces of $\mathcal{E}_2$ are observable in $\mathcal{E}_5$, but $\mathcal{E}_5$ accepts more behaviors since the action*

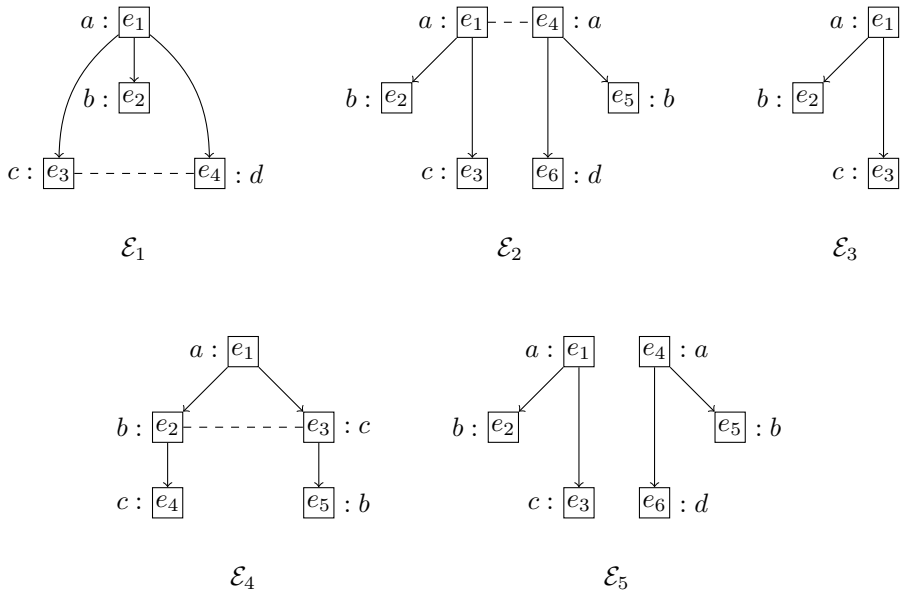



Figure A.1: Conformance Relations with Symmetric Interaction.

*a is possible twice concurrently, so* $\mathcal{E}_2 \leq_{tr} \mathcal{E}_5$*, but* $\neg(\mathcal{E}_5 \leq_{tr} \mathcal{E}_2)$*. Both* $\mathcal{E}_3, \mathcal{E}_4$ *allow the same actions, but one of them in a sequential way while the other does it concurrently, we have then* $\neg(\mathcal{E}_3 \leq_{tr} \mathcal{E}_4)$ *and* $\neg(\mathcal{E}_4 \leq_{tr} \mathcal{E}_3)$*. When interleaving semantics are considered, sequential traces of both* $\mathcal{E}_3$ *and* $\mathcal{E}_4$ *coincide, thus* $\mathcal{E}_3 \leq^s_{tr} \mathcal{E}_4$ *and* $\mathcal{E}_4 \leq^s_{tr} \mathcal{E}_3$*. The same is also true for* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *as their traces coincide.*

With both semantics, $\mathcal{E}_2$ correctly implements $\mathcal{E}_1$ w.r.t trace preorder, but $\mathcal{E}_1$ specifies that after $a$ there is a choice between $b$ and $c$, while $\mathcal{E}_2$ may refuse one of these. The reason of this is that both relations only consider sequences (respectively partial order) of actions as observations, and not whether conflicts are resolved internally by the system, or externally by the environment.

## A.3 Testing Preorder

In order to have a stronger relation that refines trace preorder, we define the set of actions that a system may refuse following [ADF86, De 87]. Under partial order semantics, we are interested not in single actions the system can refuse, but in sets of concurrent actions (where all possible concurrent sets of actions are denoted by $\mathcal{CO}(L)$). Since we will use **refuses** with **after** and we allow nondeterminism in the system (the reached configuration may not be unique), we extend the definition of refusals to sets of configurations.

**Definition** A.4 (Refusals). *Let* $\mathcal{E} \in \mathcal{ES}(L)$ *with* $C \in \mathcal{C}(\mathcal{E}), S \subseteq \mathcal{C}(\mathcal{E})$ *and* $A_1 \subseteq$

*$L, A_2 \subseteq \mathcal{CO}(L)$, we define*

$$\begin{array}{rcl} C \ \textbf{\textit{refuses}}_s \ A_1 & \triangleq & \forall a \in A_1 : C \stackrel{a}{\not\Longrightarrow}_s \\ S \ \textbf{\textit{refuses}}_s \ A_1 & \triangleq & \exists C \in S : C \ \textbf{\textit{refuses}}_s \ A_1 \\ C \ \textbf{\textit{refuses}} \ A_2 & \triangleq & \forall \omega \in A_2 : C \stackrel{\omega}{\not\Longrightarrow} \\ S \ \textbf{\textit{refuses}} \ A_2 & \triangleq & \exists C \in S : C \ \textbf{\textit{refuses}} \ A_2 \end{array}$$

In addition to requiring that any trace of the implementation is allowed in the specification, we require that any time the implementation refuses to perform a new action (respectively a set of concurrent actions), this is also the case in the specification.

**Definition** A.5 (Testing Preorder). *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and implementation of the system, then*

$$\begin{array}{rcl} \mathcal{I} \leq_{te}^{s} \mathcal{S} & \Leftrightarrow & \forall \sigma \in L^*, A_1 \subseteq L : \\ & & (\mathcal{I} \ \textbf{\textit{after}}_s \ \sigma) \ \textbf{\textit{refuses}}_s \ A_1 \Rightarrow (\mathcal{S} \ \textbf{\textit{after}}_s \ \sigma) \ \textbf{\textit{refuses}}_s \ A_1 \end{array}$$

$$\begin{array}{rcl} \mathcal{I} \leq_{te} \mathcal{S} & \Leftrightarrow & \forall \omega \in \mathcal{PS}(L), A_2 \subseteq \mathcal{CO}(L) : \\ & & (\mathcal{I} \ \textbf{\textit{after}} \ \omega) \ \textbf{\textit{refuses}} \ A_2 \Rightarrow (\mathcal{S} \ \textbf{\textit{after}} \ \omega) \ \textbf{\textit{refuses}} \ A_2 \end{array}$$

Example A.2 (Testing Preorder). *Consider again Figure A.1. Under interleaving semantics, traces and refusals of $\mathcal{E}_3$ and $\mathcal{E}_4$ coincide, therefore $\mathcal{E}_3 \leq_{te} \mathcal{E}_4$ and $\mathcal{E}_4 \leq_{te} \mathcal{E}_3$. From the initial configuration, $\mathcal{E}_1$ refuses $\{b, c, d\}$ and this is also the case in $\mathcal{E}_2$. As after $a$, $\mathcal{E}_1$ only refuses $\{a\}$ and this is also the case in $\mathcal{E}_2$, we have $\mathcal{E}_1 \leq_{te} \mathcal{E}_2$. The converse is not true; after $a$, system $\mathcal{E}_2$ can refuse $d$ if, for example, it follows the left branch. This action is not refused by $\mathcal{E}_1$ and therefore $\neg(\mathcal{E}_2 \leq_{te} \mathcal{E}_1)$. System $\mathcal{E}_3$ also refuses $d$ after $a$ and $\neg(\mathcal{E}_3 \leq_{te} \mathcal{E}_1)$. Note that $\leq_{te}$ does not allow extra traces in the implementation. In fact $\neg(\mathcal{E}_1 \leq_{te} \mathcal{E}_3)$ since $(\mathcal{E}_1 \ \textbf{\textit{after}} \ a \cdot d) \ \textbf{\textit{refuses}} \ \emptyset$, yet $(\mathcal{E}_3 \ \textbf{\textit{after}} \ a \cdot d) = \emptyset$, hence $\neg((\mathcal{E}_3 \ \textbf{\textit{after}} \ a \cdot d) \ \textbf{\textit{refuses}} \ \emptyset)$.*

## A.4 Relaxing Testing Preorder

A practical modification to testing preorder is to restrict all the traces to only the ones contained in the specification [Bri88]. This relation requires that the implementation does what it has to do, not that it does not what it is not allowed to do. It allows underspecification, i.e. only a subset of the functionalities of the actual system are specified and implementations with extra behaviors are considered correct.

**Definition** A.6 (The **conf** relation). *Let $\mathcal{S}, \mathcal{I} \in \mathcal{ES}(L)$ be respectively the specification and implementation of the system, then*

$$\begin{array}{rcl} \mathcal{I} \ \textbf{\textit{conf}}_s \ \mathcal{S} & \Leftrightarrow & \forall \sigma \in \mathit{traces}_s(\mathcal{S}), A_1 \subseteq L : \\ & & (\mathcal{I} \ \textbf{\textit{after}}_s \ \sigma) \ \textbf{\textit{refuses}}_s \ A_1 \Rightarrow (\mathcal{S} \ \textbf{\textit{after}}_s \ \sigma) \ \textbf{\textit{refuses}}_s \ A_1 \end{array}$$

$$\begin{array}{rcl} \mathcal{I} \ \textbf{\textit{conf}} \ \mathcal{S} & \Leftrightarrow & \forall \omega \in \mathit{traces}(\mathcal{S}), A_2 \subseteq \mathcal{CO}(L) : \\ & & (\mathcal{I} \ \textbf{\textit{after}} \ \omega) \ \textbf{\textit{refuses}} \ A_2 \Rightarrow (\mathcal{S} \ \textbf{\textit{after}} \ \omega) \ \textbf{\textit{refuses}} \ A_2 \end{array}$$

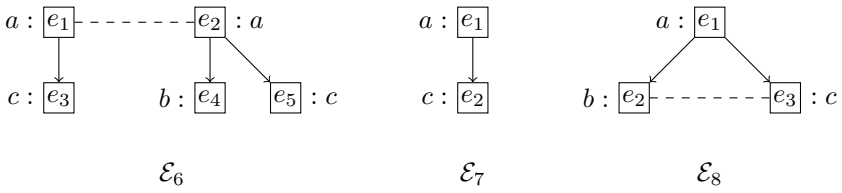



Figure A.2: The **conf** relation is not transitive.

Example A.3 (The **conf** relation). *We saw in Example A.2 that* $\mathcal{E}_1 \leq_{te} \mathcal{E}_2$, *and since* **conf** *considers the traces of* $\mathcal{E}_2$ *only, we have* $\mathcal{E}_1$ ***conf*** $\mathcal{E}_2$. *Since the relation* **conf** *is based on the traces of the specification only, it allows extra behaviors in the implementation. So even if* $\neg(\mathcal{E}_3 \leq_{te} \mathcal{E}_1)$, *we have* $\mathcal{E}_3$ ***conf*** $\mathcal{E}_1$.

The following result relates the different implementation relations under partial order semantics.

**Proposition** 8. *The conformance relations based on symmetric interaction have the following properties*

1. $\leq_{tr}$ *and* $\leq_{te}$ *are preorders;* **conf** *is reflexive.*
2. $\leq_{te} = \leq_{tr} \cap$ **conf**

*Proof.* Point 1 being obvious, we only show point 2, by proving that the inclusion holds in both directions. Suppose $\mathcal{I} \not\leq_{tr} \mathcal{S}$, then there exists $\omega \in \mathcal{LPO}(L)$ such that $\mathcal{I} \stackrel{\omega}{\Longrightarrow}$, but $\mathcal{S} \not\stackrel{\omega}{\Longrightarrow}$, thus $(\mathcal{S}$ **after** $\omega) = \emptyset$ and $\neg((\mathcal{S}$ **after** $\omega)$ **refuses** $\emptyset)$ while $(\mathcal{I}$ **after** $\omega)$ **refuses** $\emptyset$, $\mathcal{I} \not\leq_{te} \mathcal{S}$ and finally $\leq_{te} \subseteq \leq_{tr}$. As **conf** is a restriction of $\leq_{te}$ to the traces of $\mathcal{S}$, it follows that $\leq_{te} \subseteq$ **conf**. Suppose $\mathcal{I} \not\leq_{te} \mathcal{S}$, then there exist $\omega \in \mathcal{LPO}(L), A \subseteq \mathcal{CO}(L)$ such that $(\mathcal{I}$ **after** $\omega)$ **refuses** $A$ and $\neg((\mathcal{S}$ **after** $\omega)$ **refuses** $A)$. If $\omega \in$ traces$(\mathcal{S})$ we have that $\neg(\mathcal{I}$ **conf** $\mathcal{S})$. If $\omega \notin$ traces$(\mathcal{S})$, we know by $(\mathcal{I}$ **after** $\omega)$ **refuses** $A$ that $\omega \in$ traces$(\mathcal{I})$ and therefore $\mathcal{I} \not\leq_{tr} \mathcal{S}$. □

Example A.4 (The **conf** relation is not transitive). *Consider Figure A.2, we have* $\mathcal{E}_7$ **conf** $\mathcal{E}_6$*: if we denote* $\{a, b, c\}$ *by* $L$, *we have* $(\mathcal{E}_7$ ***after*** $a)$ ***refuses*** $\{a, b\}$ *and* $(\mathcal{E}_6$ ***after*** $a)$ ***refuses*** $\{a, b\}$*; we also have* $(\mathcal{E}_7$ ***after*** $a \cdot c)$ ***refuses*** $L$ *and* $(\mathcal{E}_6$ ***after*** $a \cdot c)$ ***refuses*** $L$*; finally* $(\mathcal{E}_7$ ***after*** $a \cdot b)$ ***refuses*** $S$ *is false for any set* $S$. *We also have* $\mathcal{E}_8$ **conf** $\mathcal{E}_7$ *since* $\mathcal{E}_8$ *has the same behavior that* $\mathcal{E}_7$ *with an additional branch. Nevertheless* $\neg(\mathcal{E}_8$ **conf** $\mathcal{E}_6)$ *since* $(\mathcal{E}_8$ ***after*** $a \cdot b)$ ***refuses*** $\{c\}$, *but* $\neg((\mathcal{E}_6$ ***after*** $a \cdot b)$ ***refuses*** $\{c\})$. *This shows that* **conf** *is not transitive.*

## A.5 Conclusion

Three conformance relations over labeled transition systems (trace preorder, testing preorder and **conf**) have been extended in this chapter to concurrency-enabled relations over labeled event structures. With the interleaving semantics,

the relations we obtain boil down to the same relations defined for LTS, since they focus on sequences of actions. The only advantage of using labeled event structures as a specification formalism for testing remains in the conciseness of the concurrent model with respect to a sequential one. As far as testing is concerned, the benefit is low since every interleaving has to be tested. By contrast, under the partial order semantics, the relations we obtain allow to distinguish explicitly implementations where concurrent actions are implemented as independent, from those where they are interleaved, i.e. implemented sequentially. Therefore, these relations are of interest when designing distributed systems, since the natural concurrency between actions that are performed in parallel by different processes can be taken into account. In particular, the fact of being unable to control or observe the order between actions taking place on different processes is not considered as an impediment anymore.

# B Tool and Experiments

In Chapter 5 we described how to obtain prefixes of the unfolding of the specification based on different testing criteria and how to construct a test suite from those prefixes using a SAT encoding. Additionally, Chapter 6 explains how to obtain a Petri net from a network of automata. Implementations of these methods have been made and are publicly available under

http://www.lsv.ens-cachan.fr/~ponce/tours

In this chapter we report on these implementations and compare the obtained test suites with the test suites generated by **ioco**; for this we use benchmarks coming from a parametric elevator example. The content of this chapter have been made during the summer internship of Konstantinos Athanasiou which I co-supervised with Stefan Scwhoon.

## B.1 The TOURS Prototype

The TOURS (Testing On Unfolded Reactive Systems) prototype contains the following implementations that are used to run our examples:

- cfa2pep
- mole
- enum

The cfa2pep script takes as an input a network of automata in the fca format [SSE03, GMAP95] and it constructs a ll_net file (equivalent to the PEP format [Gra95]) which is given as an input to the unfolding algorithm. Our test case generation method is based on the MOLE unfolding tool [Sch] which constructs a complete prefix of the unfolding based on the cut-off criterion presented in [ERV02] where an event is marked as a cut-off if the marking it generates

is already in the prefix, but not necessarily on the same branch. This cut-off criterion generates a test suite covering all-states and all-transitions, however, as explained in Example 5.21, the test suite obtained from this prefix usually detects less incorrect implementations. We add the inclusion criterion to the MOLE tool which can be called running the command

```
$ mole [-inc] [-e] file.ll_net
```

The tool uses by default the cut-off notion of [ERV02] and produces as an output the unfolding prefix as a Petri net (file.mci) and the SAT formula of Section 5.2.5 (file.cl). The "-inc" flag forces the tool to use the inclusion criterion and the "-e" flag produces in addition of the Petri net prefix, its equivalent event structure (file_es.dot). The SAT formula and the event structure are given as input to the enum script which calls the Z3 solver [MB08] to find all the solutions of the formula: each solution corresponds to a global test case (labels of the events in the file_es.dot file display the test cases to which the event belongs).

To run an example, execute

```
$ perl cfa2pep.pl file.fsa > file.ll_net
$ mole -inc -e file.ll_net
$ python enum.py file.cl file_es.dot
```

We have run our prototype for the agency example using the [ERV02] criterion. When using partial order semantics, the obtained prefix has 12 events and generates 2 global test cases, while using interleaving semantics we obtain a complete test graph with 330 transitions that generates 36 different test cases. In Section B.3 we run experiments on a parametric example that allows to exploit the exponential reduction in computational time when partial orders are used.

## B.2 The Elevator Example

Next section presents experiment's results based on a parametric elevator example which serves calls at $n$ different floors with $m$ elevators. This section illustrates the example for two floors and one elevator. We present the example as a network of automata (which is equivalently captured by a Petri net as it is shown in Section 6.2), unfold it with the inclusion criterion and use the SAT encoding to construct the test suite; this is done by the TOURS prototype.

The behavior of the system is distributed in the following components or processes which are represented by the automata of Figure B.1:

**Floors:** each floor consists on a button that can be pressed to call an elevator. The floor is in an *idle* state where the elevator can be called ($?call_i$), after this, it sends the call to the controllers of every elevator $e_j$ ($e_j$-*takes-call*$_i$) followed by a synchronization action that the door of elevator $e_j$ has been opened at that floor ($e_j$-*opened-at-*$f_i$), returning to the *idle* state.

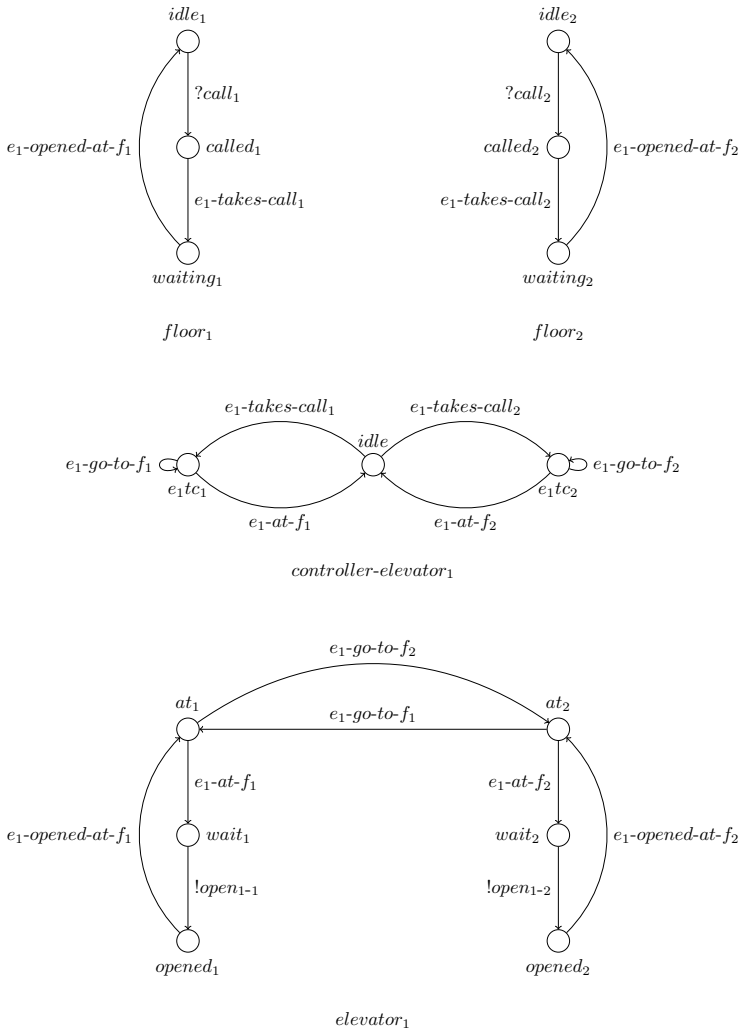



Figure B.1: Network of automata of the elevator example with one elevator and two floors.

**Controllers of elevators:** the controller of each elevator $e_j$ starts at an *idle* state and it can take the call from any floor $f_i$. From here, the controller can either move the elevator to the corresponding floor ($e_j$-*go-to*-$f_i$) or acknowledge that the elevator is already at that floor ($e_j$-*at*-$f_i$).

**Elevators:** each elevator starts at some floor, i.e. state $at_i$. From this state it can tell its controller that it is already on the floor or it can move to another floor. When the elevator is in floor $f_i$, it opens the door ($!open_{j\text{-}i}$) and acknowledge this action to the corresponding floor.

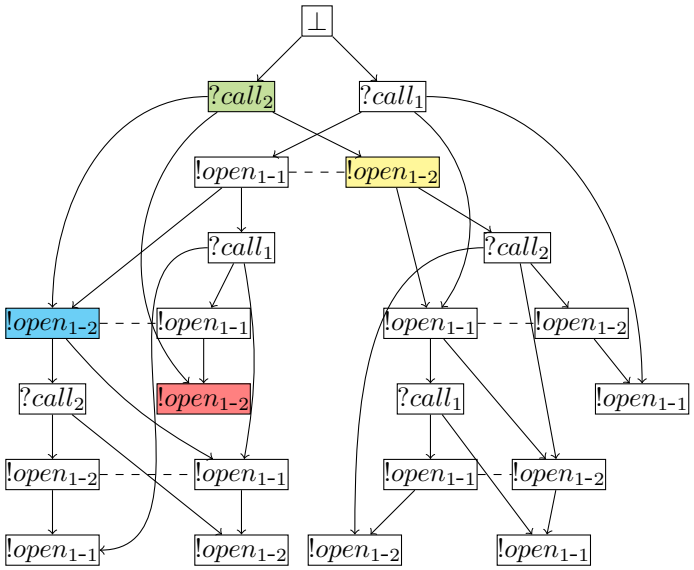



Figure B.2: Global test case of the elevator example obtained by TOURS and the inclusion criterion.

This system is given as an input to the unfolding algorithm (using the inclusion criterion) and returns a prefix which observable behavior is captured by the event structure of Figure B.2 ($\tau$ actions were removed for the presentation of the example). This prefix has not immediate conflict between inputs, therefore the SAT encoding has a unique solution representing the same prefix which is the only global test case generated by our method.

We expect that every call at any floor is eventually served (the door eventually opens at that floor) in any correct implementation. Consider the test case of Figure B.2 and the $?call_2$ action displayed in green, this call is followed by an $!open_{1\text{-}2}$ action in any maximal configuration. The yellow $!open_{1\text{-}2}$ corresponds to the scenario where the call is immediately served; the blue $!open_{1\text{-}2}$ reflects the fact that the elevator can be called concurrently at other floor ($?call_1$) and this call can be served before ($!open_{1\text{-}1} \leq !open_{1\text{-}2}$); the red $!open_{1\text{-}2}$ shows that two calls from the first floor can be served before serving the call at the second one. The latter shows that there are not priorities between serving at different floors, however all the calls are eventually served. A similar analysis can be made for the other call actions.

## B.3 Experiments

In this section we run experiments for different numbers of floors and elevators to compare our results with the test suites obtained by the **ioco** theory. We compare not only the size of the test cases but also the size of the test suite.

### B.3.1 Adding Floors and Elevators

The example presented in Section B.2 can easily be parametrized to add floors and elevators. If a new floor $f_i$ is added, in addition of adding a new automaton for the floor with transitions $e_j\text{-}takes\text{-}call_i$ for each elevator $e_j$, the existing automata representing elevators and controllers need to be extended: a new state $e_jtc_i$ is added to the controller of every elevator $e_j$ with transitions

$$idle \xrightarrow{e_j\text{-}takes\text{-}call_i} e_jtc_i$$
$$e_jtc_i \xrightarrow{e_j\text{-}go\text{-}to\text{-}f_i} e_jtc_i$$
$$e_jtc_i \xrightarrow{e_j\text{-}at\text{-}f_i} idle$$

In addition, the states $at_i, wait_i$ and $opened_i$ are added to the elevator $e_j$ with transitions

$$at_i \xrightarrow{e_j\text{-}at\text{-}f_i} wait_i$$
$$wait_i \xrightarrow{!open_{j\text{-}i}} opened_i$$
$$opened_i \xrightarrow{e_j\text{-}opened\text{-}at\text{-}f_i} at_i$$

and for each exiting floor $k < i$ all the possible movements between them and the new floor are added, i.e.

$$at_k \xrightarrow{e_j\text{-}go\text{-}to\text{-}f_i} at_i$$
$$at_i \xrightarrow{e_j\text{-}go\text{-}to\text{-}f_k} at_k$$

If a new elevator is added, two automata (representing the elevator itself and its controller) are added and for every floor $f_i$ we add the possibility that the new elevator $e_j$ serves its call, i.e. we add transitions

$$called_i \xrightarrow{e_j\text{-}takes\text{-}call_i} waiting_i$$

### B.3.2 Setting Up the Experiments

In order to make a fair comparison of the algorithms presented in this thesis and the algorithms of the **ioco** theory, we need to use the same test selection method. Since available tool such as TGV [JJ05] or JtorX [Bel10] use test purposes rather than a testing criterion, we propose to obtain the complete test graph (CTG) by our method converting the LTS into a PN without concurrency. Since we do not complete the specification with underspecified outputs, we only obtain an approximation of the CTG which gives us a lower bound on its number of events. Once we obtain our CTG and since concurrency between inputs and outputs is converted into conflict between all the possible interleavings, we solve controllability problems by an extended version of the SAT encoding in Section 5.2.5 where we allow conflicts between inputs and outputs. Since

| Floors | Elevators | Prefix | GT | Time | CTG | LT | Time |
|---|---|---|---|---|---|---|---|
| 2 | 1 | 11 | 1 | 1 | 95 | 14 | 7 |
| 2 | 2 | 29 | 1 | 1 | 3929 | ✗ | > 86400 |
| 3 | 1 | 43 | 1 | 1 | 2299 | ✗ | > 86400 |
| 3 | 2 | 220 | 1 | 6 | 3911179 | ✗ | > 86400 |
| 3 | 3 | 1231 | 1 | 144 | ✘ | ✗ | > 86400 |
| 4 | 1 | 219 | 1 | 14 | ✘ | ✗ | > 86400 |
| 4 | 2 | 1853 | 1 | 291 | ✘ | ✗ | > 86400 |
| 4 | 3 | 17033 | 1 | 12800 | ✘ | ✗ | > 86400 |
| 4 | 4 | 140873 | ✗ | > 86400 | ✘ | ✗ | > 86400 |

Table B.1: Experiments.

the latter only solves controllability problem, it is equivalent to the backtrack strategy used by TGV.

When using the inclusion criterion, the number of events in the complete test graph becomes too big for very simple examples and the SAT solver takes too much computational time to compute all the test cases. For this reason, we decided to run the experiments using the original cut-off criteria of MOLE which assures transition and state coverage.

### B.3.3 Results

Table B.1 reports the number of events in the unfolding's prefix obtained by our method, the number of global test cases seen as event structures (GT), the number of transitions in the approximation of the complete test graph, the number of test cases seen as labeled transition systems (LT) and the corresponding computational times in seconds. The unfolding tool and the SAT encoding consider internal events, while the sizes displayed on the prefix and CTG column only consider observable events.

We can easily observe in the table the exponential explosion in the number of events when interleavings are used. In addition we see that it does not matter how many floors or elevators we add, since the example does not introduce conflict between inputs, the obtained global test case is always unique. In contrast to this, the number of local test cases increases in the interleaving setting since concurrency is transformed into conflict that needs to be solved to avoid controllability problems.

The blue ✘ symbol indicates that the unfolding tool was not able to obtain a finite prefix (complete test graph) after more than twenty-four hours, while the red ✗ symbol indicates that the SAT solver was not able to find solutions after more than twenty-four hours (for more than 3 floors and 2 elevator, we were not able to run the SAT solver since the unfolding was not finished).

The unfolding of the net for 3 floors - 2 elevators example using interleaving semantics (when internal actions are considered) contains 15353982 events, showing that the unfolding tool can handle very big examples. However, the SAT solver was not able to handle encodings with more than 84820 variables (net for the 4 floors - 3 elevators example). Since causality is transitive and con-

flict is inherited w.r.t causal dependence, the SAT encoding can be improved by just considering observable events, however immediate causality and immediate conflict between only observable events need to be computed increasing again the computational time of the method. We are currently working on the implementation to achieve a better performance by just considering observable events.

# Bibliography


[Abr87] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53:225–241, 1987.

[ACRR10] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Analysing unlinkability and anonymity using the applied pi calculus. In *Computer Security Foundations Symposium*, pages 107–121. IEEE Computer Society, 2010.

[ADF86] Luca Aceto, Rocco De Nicola, and Alessandro Fantechi. Testing equivalences for event structures. In *Mathematical Models for the Semantics of Parallelism*, volume 280 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 1986.

[Alb76] David S. Alberts. The economics of software quality assurance. In *National Computer Conference and Exposition*, volume 45 of *AFIPS Conference Proceedings*, pages 433–442. AFIPS Press, 1976.

[AO08] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[Bel10] Axel Belinfante. JTorX: A tool for on-line model-driven test derivation and execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270, 2010.

[Ber91] Gilles Bernot. Testing against formal specifications: A theoretical view. In *International Conference on Theory and Practice of Software Development*, volume 494 of *Lecture Notes in Computer Science*, pages 99–119. Springer, 1991.

[BGMK07] Puneet Bhateja, Paul Gastin, Madhavan Mukund, and K. Narayan Kumar. Local testing of message sequence charts is difficult. In *International Symposium on Fundamentals of Computation Theory*, volume 4639 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2007.

[BJSK12] Nathalie Bertrand, Thierry Jéron, Amélie Stainer, and Moez Krichen. Off-line test selection with test purposes for non-deterministic timed automata. *Logical Methods in Computer Science*, 8(4), 2012.

[BK85] Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:77–121, 1985.

[BK96] Falko Bause and Pieter S. Kritzinger. *Stochastic Petri nets - an introduction to the theory*. Advanced studies of computer science. Vieweg, 1996.

[BM08] Puneet Bhateja and Madhavan Mukund. Tagging make local testing of message-passing systems feasible. In *International Conference on Software Engineering and Formal Methods*, pages 171–180. IEEE Computer Society, 2008.

[BR70] John N. Buxton and Brian Randell, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division*. NATO Science Committee, 1970.

[Bri88] Ed Brinksma. A theory for the derivation of tests. In *Protocol Specification, Testing and Verification*, pages 63–74. North-Holland, 1988.

[BW90] Jos C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[CFP96] Ana R. Cavalli, Jean Philippe Favreau, and Marc Phalippou. Standardization of formal methods in conformance testing of communication protocols. *Computer Networks and ISDN Systems*, 29(1):3–14, 1996.

[De 87] Rocco De Nicola. Extensional equivalences for transition systems. *Acta Informatica*, 24(2):211–237, 1987.

[DH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[DR95] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces*. World Scientific Publishing Co., Inc., 1995.

[ERV02] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.

[Fab06] Éric Fabre. On the construction of pullbacks for safe Petri nets. In *International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, volume 4024 of *Lecture Notes in Computer Science*, pages 166–180. Springer, 2006.

[FGGT08] Alain Faivre, Christophe Gaston, Pascale Le Gall, and Assia Touil. Test purpose concretization through symbolic action refinement. In *International Conference on Testing Communicating Systems / Workshop on Formal Approaches to Testing of Software*, volume 5047 of *Lecture Notes in Computer Science*, pages 184–199. Springer, 2008.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Australian Computer Science Conference*, pages 56–66, 1988.

[Gar05] Simson Garfinkel. History's worst software bugs. http://archive.wired.com/software/coolapps/news/2005/11/69355, 2005. Visited on 31-8-2014.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 82–96. Springer, 1995.

[GG99] Stephane Gaubert and Alessandro Giua. Petri net languages and infinite subsets of $\mathbb{N}^m$. *Journal of Computer and System Sciences*, 59(3):373–391, 1999.

[GGRT06] Christophe Gaston, Pascale Le Gall, Nicolas Rapin, and Assia Touil. Symbolic execution techniques for test purpose definition. In *International Conference on Testing Communicating Systems*, volume 3964 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2006.

[GL03] Sahika Genc and Stéphane Lafortune. 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*, pages 316–336. Springer, 2003.

[GMAP95] Bernd Grahlmann, Matthias Moeller, Ulrich Anhalt, and Marienburger Platz. A new interface for the PEP tool - parallel finite automata. In *Workshop Algorithmen und Werkzeuge fur Petrinetze*, pages 21–26, 1995.

[Gra95] Bernd Grahlmann. PEP: A programming environment based on Petri nets. *Application and Theory of Petri Nets, Tool Presentation*, pages 1–6, 1995.

[GS04] Christophe Gaston and Dirk Seifert. Evaluating coverage based testing. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 293–322. Springer, 2004.

[Hen88] Matthew Hennessy. *Algebraic theory of processes*. MIT Press series in the foundations of computing. MIT Press, 1988.

[Hen97] Olaf Henniger. On test case generation from asynchronously communicating state machines. In *International Workshop on Testing Communicating Systems*, IFIP Conference Proceedings, pages 255–271. Springer, 1997.

[Hie97] Robert M. Hierons. Testing from a finite-state machine: Extending invertibility to sequences. *The Computer Journal*, 40(4):220–230, 1997.

[HJJ07] Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing input/output partial order automata. In *International Conference on Testing Communicating Systems / Workshop on Formal Approaches to Testing of Software*, volume 4581 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2007.

[HLM+08] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008.

[HMN08] Robert M. Hierons, Mercedes G. Merayo, and Manuel Núñez. Implementation relations for the distributed test architecture. In *International Conference on Testing Communicating Systems / Workshop on Formal Approaches to Testing of Software*, volume 5047 of *Lecture Notes in Computer Science*, pages 200–215. Springer, 2008.

[HMN12] Robert M. Hierons, Mercedes G. Merayo, and Manuel Núñez. Using time to add order to distributed testing. In *International Symposium on Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2012.

[Hoa85] Charles A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HT97] Lex Heerink and Jan Tretmans. 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*, pages 23–38. Chapman & Hall, 1997.

[HT03] David Harel and P.S Thiagarajan. Message Sequence Charts. In *UML for Real: Design of Embedded Real-Time Systems*, pages 75–105. Kluwer Academic Publishers, 2003.

[HU08] Robert M. Hierons and Hasan Ural. The effect of the distributed test architecture on the power of testing. *The Computer Journal*, 51(4):497–510, 2008.

[ISO89] ISO 8807. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989.

[Jar03] Claude Jard. Synthesis of distributed testers from true-concurrency models of reactive systems. *Information & Software Technology*, 45(12):805–814, 2003.

[Jér09] Thierry Jéron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240:167–184, 2009.

[JJ05] Claude Jard and Thierry Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4):297–315, 2005.

[JJKV98] Claude Jard, Thierry Jéron, Hakim Kahlouche, and César Viho. Towards automatic distribution of testers for distributed conformance testing. In *Formal Techniques for Networked and Distributed Systems*, volume 135 of *IFIP Conference Proceedings*, pages 353–368. Kluwer, 1998.

[JJTV99] Claude Jard, Thierry Jéron, Lénaick Tanguy, and César Viho. Remote testing can be as powerful as local testing. In *Formal Techniques for Networked and Distributed Systems*, volume 156 of *IFIP Conference Proceedings*, pages 25–40. Kluwer, 1999.

[JM99] Thierry Jéron and Pierre Morel. Test generation derived from model-checking. In *International Conference on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 108–121. Springer, 1999.

[Kat99] Joost-Pieter Katoen. Concepts, algorithms, and tools for model checking, 1999. Lecture notes. Universität Erlangen-Nürnberg.

[KK97] Sungwon Kang and Myungchul Kim. Interoperability test suite derivation for symmetric communication protocols. In *Formal Techniques for Networked and Distributed Systems*, volume 107 of *IFIP Conference Proceedings*, pages 57–72. Chapman & Hall, 1997.

[KKKV06] Victor Khomenko, Alex Kondratyev, Maciej Koutny, and Walter Vogler. Merged processes: a new condensed representation of Petri net behaviour. *Acta Informatica*, 43(5):307–330, 2006.

[KM02] Dietrich Kuske and Rémi Morin. Pomsets for local trace languages. *Journal of Automata, Languages and Combinatorics*, 7(2):187–224, 2002.

[KT09] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[LG05] Grégory Lestiennes and Marie-Claude Gaudel. Test de systèmes réactifs non réceptifs. *Journal Européen des Systèmes automatisés*, 39(1-3):255–270, 2005.

[Lon12] Delphine Longuet. Global and local testing from message sequence charts. In *Symposium on Applied Computing*, pages 1332–1338. ACM, 2012.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123, 1996.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland / Elsevier, 1989.

[Maz88] Antoni W. Mazurkiewicz. Basic notions of trace theory. In *Workshop on Research and Education in Concurrent Systems*, volume 354 of *Lecture Notes in Computer Science*, pages 285–363. Springer, 1988.

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[McM95] Kenneth L. McMillan. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, 1995.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[Mil89] Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1201–1242. MIT Press, 1990.

[MY10] Andrey Mokhov and Alexandre Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[Mye04] Glenford J. Myers. *The art of software testing (2nd. ed.)*. Wiley, 2004.

[NPW81] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theoretical Computer Science*, 13:85–108, 1981.

[Par81] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Universitt Hamburg, 1962.

[Phi87] Iain Phillips. Refusal testing. *Theoretical Computer Science*, 50:241–284, 1987.

[PHL12] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Conformance relations for labeled event structures. In *International Conference on Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2012.

[PHL13] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Unfolding-based test selection for concurrent conformance. In *International Conference on Testing Software and Systems*, volume 8254 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2013.

[PHL14a] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Distributed testing of concurrent systems: vector clocks to the rescue. In *International Colloquium on Theoretical Aspects of Computing*, volume 8687 of *Lecture Notes in Computer Science*, pages 369–387. Springer, 2014.

[PHL14b] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Model-based testing for concurrent systems: Unfolding-based test selection. *International Journal on Software Tools for Technology Transfer*, 2014. To appear.

[PHL14c] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Model-based testing for concurrent systems with labeled event structures. *Software Testing, Verification and Reliability*, 24(7):558–590, 2014.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004.

[PM14] Hernán Ponce de León and Andrey Mokhov. Building bridges between sets of partial orders. http://hal.inria.fr/hal-01060449, 2014. Technical report. Visited on 4/9/2014.

[PS96] Jan Peleska and Michael Siegel. From testing theory to test driver implementation. In *International Symposium of Formal Methods Europe*, volume 1051 of *Lecture Notes in Computer Science*, pages 538–556. Springer, 1996.

[PZ13] Louchka Popova-Zeugmann. *Time and Petri Nets*. Springer, 2013.

[Rod13] César Rodríguez. *Verification Based on Unfoldings of Petri Nets with Read Arcs*. Thèse de doctorat, Laboratoire Spécification et Vérification, ENS Cachan, France, December 2013.

[RS97] Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of formal languages, vol. 3: beyond words*. Springer-Verlag New York, Inc., 1997.

[Sch] Stefan Schwoon. The MOLE unfolding tool. http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/. Visited on 31/8/2014.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., 1st edition, 1999.

[Seg97] Roberto Segala. Quiescence, fairness, testing, and the notion of implementation. *Information and Computation*, 138(2):194–210, 1997.

[SNW96] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, 1996.

[SSE03] Claus Schröter, Stefan Schwoon, and Javier Esparza. The model-checking kit. In *International Conference on Applications and Theory of Petri Nets*, volume 2679 of *Lecture Notes in Computer Science*, pages 463–472. Springer, 2003.

[STS13] Willem Gerrit Johan Stokkink, Mark Timmer, and Mariëlle Stoelinga. Divergent quiescent transition systems. In *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 214–231. Springer, 2013.

[Tre92] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.

[Tre96a] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[Tre96b] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

[Tre08] Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008.

[UK97] Andreas Ulrich and Hartmut König. Specification-based testing of concurrent systems. In *Formal Techniques for Networked and Distributed Systems*, volume 107 of *IFIP Conference Proceedings*, pages 7–22. Chapman & Hall, 1997.

[UK99] Andreas Ulrich and Hartmut König. Architectures for testing distributed systems. In *International Workshop on Testing Communicating Systems*, volume 147 of *IFIP Conference Proceedings*, pages 93–108. Kluwer, 1999.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, 2007.

[vG89] Rob J. van Glabbeek and Ursula Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In *Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 1989.

[vGGSU12] Rob J. van Glabbeek, Ursula Goltz, and Jens-Wolfhard Schicke-Uffmann. On distributability of Petri nets - (extended abstract). In *International Conference on Foundations of Software Science and Computation Structures*, volume 7213 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2012.

[vHJJ08] Gregor von Bochmann, Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing systems specified as partial order input/output automata. In *International Conference on Testing Communicating Systems / Workshop on Formal Approaches to Testing of Software*, volume 5047 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2008.

[Vog02] Walter Vogler. Partial order semantics and read arcs. *Theoretical Computer Science*, 286(1):33–63, 2002.

[Win85] Glynn Winskel. Petri nets, morphisms and compositionality. In *Applications and Theory in Petri Nets*, volume 222 of *Lecture Notes in Computer Science*, pages 453–477. Springer, 1985.

[XS09] Yang Xu and Ken S. Stevens. Automatic synthesis of computation interference constraints for relative timing verification. In *International Conference on Computer Design*, pages 16–22. IEEE, 2009.