Test Case Generation for Concurrent Systems Using Event Structures

Konstantinos Athanasiou[1⋆], Hernán Ponce-de-León[2⋆], and Stefan Schwoon[3]

[1] College of Computer and Information Science, Northeastern University
konathan@ccs.neu.edu
[2] Helsinki Institute for Information Technology HIIT and Department of Computer Science and Engineering, School of Science, Aalto University
hernan.poncedeleon@aalto.fi
[3] LSV (École Normale Supérieure de Cachan & CNRS), France
schwoon@lsv.ens-cachan.fr

Abstract. This paper deals with the test-case generation problem for concurrent systems that are specified by true-concurrency models such as Petri nets. We show that using true-concurrency models reduces both the size and the number of test cases needed for achieving certain coverage criteria. We present a test-case generation algorithm based on Petri net unfoldings and a SAT encoding for solving controllability problems in test cases. Finally, we evaluate our algorithm against traditional test-case generation methods under interleaving semantics.

1 Introduction

The aim of testing is to execute a *system under test* (SUT) on a set of input data that was selected with the aim of finding discrepancies between the actual behavior of the SUT and its intended behavior as described by some specification. Model-based testing additionally requires a behavioral description of the SUT. One of the most popular formalisms studied in model-based testing is that of *input-output labeled transition systems* (IOLTS) where the correctness (or conformance) relation that the SUT must verify w.r.t. its specification is formalized by the **ioco** relation [1]. This relation has become a standard, and it is used as a basis in several testing theories for extended state-based models [2–5].

Model-based testing then consists of three steps: (1) exploring the specification to obtain a representation of relevant behaviours to test; (2) generating a suite of test cases from the mentioned representation; and (3) applying the tests to the SUT. This paper mainly deals with step (2) in the context of concurrent systems.

In the **ioco** theory, step (1) generates a *complete test graph* $\mathcal{G}$ describing the inputs the tester may propose and the outputs the system may produce, up to a depth that fulfills a given *test purpose*. Consider the example graph in Fig. 1 (a).

⋆ This research was done while the authors were part of LSV, supported by an INRIA internship and the TECSTES project.

It specifies that every tester should begin with input i_1, to which the system ought to respond by o_1. After this, there are two choices: the tester proposes either input i_2 or input i_3, to which the system should react accordingly. If the system shows an unexpected output or no output at all, it is deemed to be non-conformant.

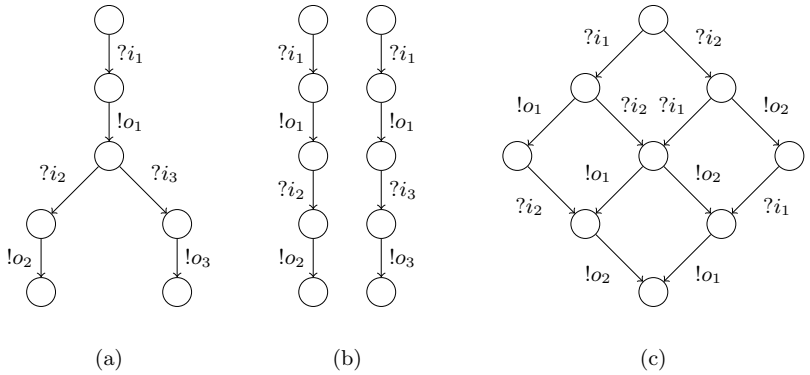


Fig. 1. Example of (a) a complete test graph; (b) the resulting test cases; (c) test graph for interleaving semantics of concurrent system.

A *test case* is a subgraph of $\mathcal{G}$ that tells the tester which inputs to choose and which outputs to expect at which point during the test. For instance, no node may have two outgoing edges labelled by inputs. The suite of test cases corresponding to $\mathcal{G}$ is obtained during step (2), e.g., by using a backtracking strategy [6]. Fig. 1 (b) shows the two test cases resulting from the graph in (a).

Model-based testing of concurrent systems has been studied in the past [7–9], but mostly in the context of interleaving semantics which suffers from state-space explosion. For instance, consider a system with two independent components C_1, C_2, where input i_k in C_k should produce output o_k, for $k = 1, 2$. Applying **ioco**-conformance methods to the interleaving semantics of this system produces the test graph in Fig. 1 (c), which in turn produces four different test cases (see Example 4).

To avoid this problem, concurrent systems can be modelled by Petri nets, whose partial-order (or true-concurrency) semantics is given by its unfolding [10]. Some of the methods originally developed for Finite State Machines [11] have been adapted to k-bounded and safe Petri nets [12, 13] while test-case generation for concurrent systems based on unfoldings has been studied in [14–16]. In particular, [16] proposed a suitable extension of **ioco** for testing concurrent systems, called **co-ioco**. In the latter, the test graph $\mathcal{G}$ is replaced by an *event structure* $\mathcal{E}$ characterizing the causal relations between inputs and outputs, and a concurrent (or global) test case becomes a prefix of $\mathcal{E}$ (with suitable properties). For instance, the example with two components C_1, C_2 leads to just one

test case, where each component receives one input and produces one output. An abstract algorithm for obtaining the suite of test cases from $\mathcal{E}$ is proposed in [16], but it is not efficient since it enumerates linearizations of $\mathcal{E}$. Moreover, in some cases it can produce the same test case several times. Also, an actual implementation of these concepts was lacking so far.

In [14] and [23] finite event structures are constructed from the specification of the system and projected into the distributed components of the system; if the projection still contains concurrency, interleaving semantics are applied. Each path of the event structure represents a test case, but they do not give an explicit algorithm to compute them and argue that optimization techniques to minimize the number of test cases is out of the scope of the paper.

More recently, unfolding techniques have been applied to test multithreaded programs [24]; their setting is different to ours since they consider a white box implementation and construct an unfolding representing the flow of the program. The constructed unfolding represents symbolic executions to avoid the explosion causes by different inputs and a SMT solver is used to generate concrete test cases.

Contributions: test-case generation based on **co-ioco** thus consists of two tasks: (i) generating a suitable prefix of $\mathcal{E}$, and (ii) extracting test cases from it. In this paper we make the following contributions:

- As for task (i), we provide a concrete implementation for generating the above-mentioned event structure $\mathcal{E}$ for certain coverage criteria, as an extension of the tool Mole [17].
- As for task (ii), we propose an improved algorithm for obtaining a test suite from $\mathcal{E}$ based on SAT-solving. The new algorithm is more efficient than [16] and does not produce the same test case several times.
- In practice, a system not only consists of inputs and outputs but also of silent transitions not observable by the tester. Naïvely adding silent events to the event structure would lead to huge test cases. We show how the test-case generation can handle silent transitions gracefully.
- Moreover, we implemented the above-mentioned components for test-case generation in a prototype tool called Tours and report on experiments. Our experiments show that keeping concurrency explicitly in the test cases not only reduces their size by avoiding interleavings, but it also reduces the number of test executions to assure a certain coverage of the system.

The paper is organised as follows: Section 2 recalls background on Petri nets, unfoldings, and test cases; Section 3 presents our theoretical contributions towards test-case generation for **co-ioco**; Section 4 discusses our implementation and experiments; we conclude in Section 6.

2 Preliminaries

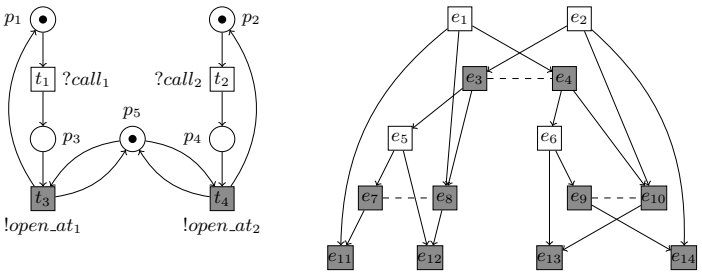
This section recalls previously known concepts used throughout the paper, such as Petri nets, event structures, and testing-related concepts. Since this paper

focuses on test-case generation from a given event structure, we focus on event structures and present the other issues more concisely. A more detailed exposition of these subjects can be found in [16].

2.1 Petri Nets and Event Structures

We deal with concurrent, reactive systems modelled as Petri nets, where we differentiate between actions proposed by the tester (*inputs*), actions produced by the system (*outputs*) and internal (*silent*) actions. We assume familiarity of the reader with Petri nets and merely recall some basic facts.

A *Petri net* consists of two disjoint finite sets P and T representing *places* and *transitions* connected by *flow arcs*. A *marking* is a distribution of *tokens* over places. In what follows, we deal with *1-safe* nets where no reachable marking places more than one token into the same place; such Petri nets can in particular represent a collection of finite automata synchronizing on common actions. Thus, we represent a marking as the set of marked places. Transitions are labeled by a mapping $\lambda : T \to \mathcal{I}n \uplus \mathcal{O}ut \uplus \{\tau\}$ over input, output and silent actions. In the following, elements of $\mathcal{I}n$ are prefixed by '?' and elements of $\mathcal{O}ut$ by '!'.



e	$\lambda(e)$	$\varphi(e)$
e_1	$?call_1$	t_1
e_2	$?call_2$	t_2
e_3	$!open\_at_2$	t_4
e_4	$!open\_at_1$	t_3
e_5	$?call_2$	t_2
e_6	$?call_1$	t_1
e_7	$!open\_at_2$	t_4
e_8	$!open\_at_1$	t_3
e_9	$!open\_at_1$	t_3
e_{10}	$!open\_at_2$	t_4
e_{11}	$!open\_at_1$	t_3
e_{12}	$!open\_at_2$	t_4
e_{13}	$!open\_at_1$	t_3
e_{14}	$!open\_at_2$	t_4

Fig. 2. A Petri net and its unfolding.

Example 1 (Petri nets). Fig. 2 (left) shows a Petri net representing an elevator serving two floors. Places are shown as circles, transitions as boxes whose shading indicates the type of their label: inputs are shown in white, outputs in grey, and silent actions in light grey (see Fig. 5). From the initial marking $\{p_1, p_2, p_5\}$, the elevator can be called concurrently at both floors (transitions t_1 and t_2); both calls can be served sequentially (t_3 or t_4), i.e. the elevator cannot open its door at both floors at the same time. This is because both transitions compete for the token at place p_5.

It is well-known (see, e.g., [10]) that a Petri net can be *unfolded* into an acyclic, potentially infinite structure that represents the partial-order semantics of the net with its possible branching behaviours. Such an unfolding directly corresponds to an *event structure* [18] :

Definition 1. *An* event structure *is a tuple* $\mathcal{E} = (E, \leq, \#, \lambda)$ *where: (i)* E *is a set of* events*; (ii)* $\leq \subseteq E \times E$ *is a partial order (called* causality*) s.t.* $\forall e \in E : |\langle e \rangle| < \infty$*, where* $\langle e \rangle = \{e' \in E \mid e' < e\}$*; (iii)* $\# \subseteq E \times E$ *is an irreflexive symmetric relation (called conflict) satisfying the property of conflict heredity, i.e.* $\forall e, e', e'' \in E : e \# e' \wedge e' \leq e'' \Rightarrow e \# e''$*; (iv) the mapping* $\lambda : E \to \mathcal{I}n \uplus \mathcal{O}ut \uplus \{\tau\}$ *labels events.*

Causality represents dependence and conflict the inability of two actions to occur together. Conflicts that are not hereditary are called *immediate*; we write $e_1 \#^i e_2$ iff for any pair (e_1', e_2') with $e_1' \leq e_1$, $e_2' \leq e_2$, $e_1' \# e_2'$ implies $e_1' = e_1$ and $e_2' = e_2$. Likewise, we consider the *immediate causality* relation $\leq^i$, where $e \leq^i f$ iff $e \leq f$ and $e \leq g \leq f$ implies $e = g$ or $g = f$. In figures, events are represented by squares, immediate causality by arrows and immediate conflict by dashed lines. 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 that are neither related by causality nor by conflict are called concurrent, i.e. $e \mathbf{\ co\ } e' \Leftrightarrow \neg(e \leq e') \wedge \neg(e \# e') \wedge \neg(e' < e)$.

An event structure derived from the unfolding of a Petri net is equipped with a function $\varphi : E \to T$ that maps events back to the net, i.e. event e is an occurrence of transition t iff $\varphi(e) = t$.

Example 2 (Unfoldings). Fig. 2 (right) shows an initial part of the unfolding of the net on the left, where the labeling is given as a table. Events represent different instances of the transitions, indicated by $\varphi(e)$ in the table. The unfolding shows that both calls can be made concurrently, i.e. $e_1 \mathbf{\ co\ } e_2$ with $\lambda(e_1) = ?call_1$ and $\lambda(e_2) = ?call_2$, but they are served sequentially, for example $e_3 \leq e_8$ with $\lambda(e_3) = !open\_at_2$ and $\lambda(e_8) = !open\_at_1$.

In an event structure, the "state" of the system is represented by the events that have occurred so far. As causality represents precedence, such a computation must be causally closed. In addition, the computation must be conflict-free.

Definition 2. *A* configuration *of* $\mathcal{E} = (E, \leq, \#, \lambda)$ *is a set* $C \subseteq E$ *such that: (i)* C *is causally closed, i.e.* $e \in C$ *implies* $\langle e \rangle \subseteq C$*; and (ii)* C *is conflict-free, i.e.* $e \in C$ *and* $e \# e'$ *imply* $e' \notin C$*. The set of configurations of* $\mathcal{E}$ *is denoted* $\mathcal{C}(\mathcal{E})$*.*

For instance, $C_1 = \{e_1, e_2, e_3\}$ is a configuration, but $\{e_1, e_2, e_3, e_4\}$ is not as e_3 conflicts with e_4. W.r.t. the original Petri net, a finite configuration represents the set of transitions that fire in some finite execution. Let $Mark(C)$ denote the marking that arises from such a firing sequence, e.g. $Mark(C_1) = \{p_2, p_3, p_5\}$.

If $\mathcal{E}$ does not contain any events labelled by τ, we call $\mathcal{E}$ *deterministic* when its configurations can be uniquely determined from its action labels, i.e. if $C \in \mathcal{C}(\mathcal{E})$ and $C \uplus \{e_1\}, C \uplus \{e_2\} \in \mathcal{C}(\mathcal{E})$, then $\lambda(e_1) = \lambda(e_2)$ implies $e_1 = e_2$. We extend

this to the case where $\mathcal{E}$ does include τ-labelled events: in that case, $\mathcal{E}$ is called deterministic if for all $C, C_1, C_2 \in \mathcal{C}(\mathcal{E})$ with $C_1, C_2 \supseteq C$, if e_1 resp. e_2 are the only non-τ-labelled events in $C_1 \setminus C$ resp. $C_2 \setminus C$, then $\lambda(e_1) = \lambda(e_2)$ implies $e_1 = e_2$.

2.2 Event Structures and Coverage Criteria

In general, the unfolding of a Petri net is infinite if the net contains a cycle of reachable markings. However, for many purposes, it suffices to study only a finite initial portion (a *prefix*) of the unfolding. With respect to testing, we are interested in finding a prefix that covers all behaviours relevant for a certain *coverage criterion* or *test purpose*. This relation was explored in [19], which proposes different criteria for truncating an unfolding w.r.t. certain coverage criteria. While the topic of coverage is somewhat orthogonal to the subject of this paper, we recall some well-known criteria:

- *all-states coverage*, i.e. every state (marking) of the specification must be covered at least once;
- *all-transition coverage*, i.e. every transition must be covered;
- *all-loops coverage*, i.e. every cycle is explored at least once.

We mention that the first two of these criteria correspond to the concept of *complete prefixes* known from the unfolding literature [10].

Definition 3. *A prefix $\mathcal{E}$ of the unfolding of a Petri net $\mathcal{N}$ is complete if for every reachable marking M of $\mathcal{N}$ there exists a configuration $C \in \mathcal{C}(\mathcal{E})$ such that: (1) $Mark(C) = M$ (i.e. M is represented in $\mathcal{E}$), and (2) for every transition t enabled in M there exists $C \uplus \{e\} \in \mathcal{C}(\mathcal{E})$ such that $\varphi(e) = t$.*

A prefix satisfying (1) but not necessarily (2) is also called *marking-complete*. For instance, the unfolding prefix in Fig. 2 (right) is complete; a prefix containing only events e_1 and e_2 would be marking-complete.

A marking-complete prefix assures all-states coverage, while a complete prefix additionally assures all-transitions coverage. A truncation method for all-loops coverage was developed in [19].

2.3 Test Cases

A test case is a specification of the tester's behavior during an experiment carried out on the system under test. A test suite is a set of test cases. During the experiment, the tester serves as an "environment" of the implementation. The tester controls the input actions and observes the output actions but does not control the latter. While the inputs made by the tester may depend on the previously observed outputs, the next input should always be uniquely determined, i.e. the tester must not have a choice between different inputs. Similarly, test cases do not contain choices between outputs and inputs, otherwise the implementation may produce an output without allowing the tester to propose the input. This

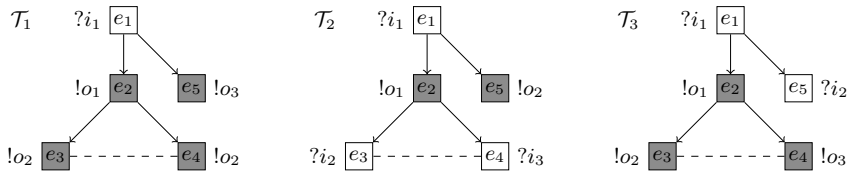


Fig. 3. Event structures as global test cases.

property is called controllability [20]. These requirements also imply that the test case is required to be deterministic. Finally, we require the experiment to terminate, therefore the test case should be finite.

In the **ioco** theory, test cases are modeled by labeled transition systems with some structural assumptions [6]: *(i)* they have an acyclic and finite structure; *(ii)* they are deterministic; *(iii)* they contain only observable actions; *(iv)* there are no choices concerning inputs; and *(v)* they are output-complete (in every state of the test case where an output is enabled, the test case must handle all the possible outputs). We refer to such objects as *sequential test cases*.

For the **co-ioco** theory, designed to handle concurrent and distributed systems, [19] proposed a different type of test case, called *global test case*. Here, the tester has control over all components in the system and can observe them all. A global test case is represented by a finite, deterministic event structure; thus, inputs and outputs in different components may happen in parallel. In addition, immediate conflict between an input and any other event is forbidden. In contrast to **ioco**, we shall also allow for silent events in the specification, since this facilitates the work of the system designer. However, these silent events are irrelevant for conducting a test and are not permitted in global test cases; see also Section 3.2. Finally, since the test execution is not modeled by parallel composition[1] as in the case of **ioco**, we drop the output-complete assumption.

It is worth to notice that 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; such local execution can be formalized as in the **ioco** case. However, a naive projection does not preserve information about concurrency; in order to make the observation of concurrency possible, further machinery is needed. An approach based on vector clocks has been proposed in [21].

Definition 4. *A* global test case *is a finite, deterministic event structure* $\mathcal{T} = (E, \leq, \#, \lambda)$ *such that (i) all events are labelled by inputs or outputs, and (ii)* $(E^{\mathcal{I}n} \times E) \cap \#^i = \emptyset$.

Example 3 (Global test cases). Fig. 3 presents three event structures. $\mathcal{T}_1$ is nondeterministic: from $\{e_1, e_2\}$ it is possible to perform $!o_2$ and reach both $\{e_1, e_2, e_3\}$

[1] See [19] for details of the test execution in the concurrent setting.

or $\{e_1, e_2, e_4\}$; $\mathcal{T}_2$ has immediate conflict between actions $?i_2$ and $?i_3$. Thus neither $\mathcal{T}_1$ nor $\mathcal{T}_2$ is a global test case. However $\mathcal{T}_3$ is finite, deterministic, and without inputs in immediate conflict, i.e. it is a global test case.

In a global test case, events are allowed to happen in parallel. With respect to sequential test cases, this has two advantages: both the size and the number of test cases can be exponentially smaller than when concurrency is represented by interleavings. First, suppose that several outputs can happen concurrently. Then, a sequential test case must consider all their orderings, meaning that its size can be exponentially larger than the size of the corresponding global test case. Secondly, suppose that several inputs can happen concurrently. In **ioco** theory, concurrency between inputs is interpreted as a nondeterministic choice between the possible interleavings, a choice that needs to be solved to avoid uncontrollability. Thus, an **ioco**-based test suite may require an exponentially larger number of test cases than in **co-ioco** to cover the same specification.

Example 4 (Global vs. sequential test cases). Consider a process calculus notation where "$\|$" is parallel composition, ";" sequentialization and "+" choices. For a specification $(i_1; o_1 \parallel i_2; o_2)$ (which is deterministic and has no choices), the test suite $\mathcal{TS}_1 = \{i_1; o_1 \parallel i_2; o_2\}$ contains one single test case to cover all the behaviors. By contrast, if interleaving semantics is used, then the test suite $\mathcal{TS}_2$ obtained by the **ioco** algorithms contains four sequential test cases:

$$\mathcal{TS}_2 = \left\{ \begin{array}{c} ?i_1; !o_1; ?i_2; !o_2 \\ ?i_2; !o_2; ?i_1; !o_1 \\ ?i_1; ?i_2; (!o_1; !o_2 + !o_2; !o_1) \\ ?i_2; ?i_1; (!o_1; !o_2 + !o_2; !o_1) \end{array} \right\}$$

3 Constructing Global Test Cases

In this section, we present a new methodology of generating test cases for **co-ioco**-conformance that offers two advantages over the methods previously presented in [16]: *(i)* it is more efficient in practice (it avoids enumerating linearizations of the causality relation); and *(ii)* it avoids generating the same test case several times.

We recall that model-based testing consists of several steps: in the first step, one obtains a representation of the behaviours that are relevant w.r.t. a given coverage criterion. In the case of **co-ioco**, this representation is an event structure $\mathcal{E}$, more precisely an unfolding prefix of the Petri net representing the specification. In a second step, which is the subject of this section, one uses $\mathcal{E}$ to obtain a suite of test cases. Since the choice of a coverage criterion is orthogonal to our subject, we henceforth assume that $\mathcal{E}$ is given. Our task then is to extract all global test cases from $\mathcal{E}$. We make the technical assumption that $\mathcal{E}$ is deterministic; note that analogous assumptions about the complete test graph are made in **ioco** settings.

A first algorithm for this purpose was presented in [16]. The algorithm takes as an input a linearization $\mathcal{L}$ of the causality relation and adds events to the

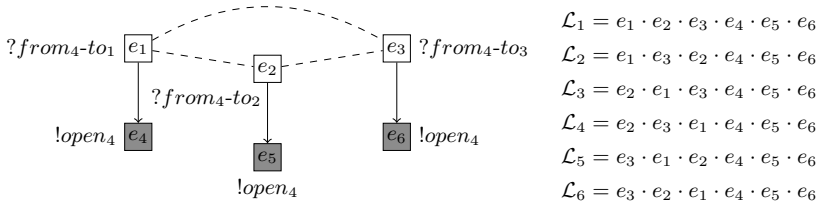


Fig. 4. A first approach for test case generation.

test case following the order of $\mathcal{L}$ whenever they do not introduce controllability problems. To obtain different test cases, the algorithm needs to be run with different linearizations. Even if using these linearizations seems to reduce the advantages of using true concurrency models, this method is only exponential on the number of immediate conflicts between inputs, which is usually very small compared with the number of all possible interleavings. However this method still has a drawback: it generates several times the same test case whenever several inputs are pairwise in immediate conflict.

Example 5 (A first approach for test case generation). Consider the event structure of Fig. 4 which represents the controller of an elevator in the 4th floor. When calling the elevator, the user indicates which floor he wants to reach. Whatever his choice, eventually the elevator arrives at the 4th floor and opens its door. According to [16], linearizations $\mathcal{L}_1$ - $\mathcal{L}_6$ are needed to construct a test suite covering the specification. If events are added one by one to the test case following these linearizations whenever they do not introduce controllability problems, $\mathcal{L}_1$ and $\mathcal{L}_2$ construct the same test case since once e_1 is added, neither e_2 nor e_3 (nor their futures) are added. This problem comes from the fact that the conflict relation consider pairs of events, but once an event is selected, the order of every other event which is in immediate conflict with it becomes irrelevant.

3.1 Encoding Test Cases by SAT

In order to solve controllability problems and avoid constructing the same test case several times as mentioned in Example 5, we propose a new, non-redundant characterization of global test cases. This characterization can be encoded in propositional logic, hence we will be able to employ a SAT solver to obtain the global test cases. Given a finite event structure, we use a SAT variable φ_e for each event e and construct a formula whose satisfying assignments correspond to global test cases. A solution assigning 1 to variable φ_e means that event e belongs to the test case, while assignment 0 means that it does not.

As test cases need to preserve causality from the specification, whenever the condition of an event is true, the conditions of its immediate causal predecessors

(and, by transitivity, all indirect precedessors) should also be true:

$$\forall e, f \in E : \bigwedge_{f \leq^i e} \varphi_e \Rightarrow \varphi_f \tag{1}$$

In addition, for each pair of immediate conflicts involving an input, at most one of them belongs to the test case (remember that immediate conflict between outputs is accepted). This is encoded as:

$$\forall e \in E, f \in E^{\mathcal{I}n} : \bigwedge_{f \#^i e} \neg\varphi_e \vee \neg\varphi_f \tag{2}$$

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 (1) or (2). An event of the prefix does not belong to the test case only if *(i)* neither does one of its immediate predecessors, or *(ii)* it is in immediate conflict with an input of the test case. We encoded this by the SAT formula

$$\forall e, f \in E, g \in E^{\mathcal{I}n} : \neg\varphi_e \Rightarrow \left(\bigvee_{f \leq^i e} \neg\varphi_f \vee \bigvee_{g \#^i e} \varphi_g \right) \tag{3}$$

Global test cases are encoded by the conjunction of (1), (2) and (3).

Example 6 (Avoiding redundancy by the SAT encoding). Consider the event structure of Fig. 4. The SAT formula of this event structure is

$$\mathrm{AMO}(\varphi_{e_1}, \varphi_{e_2}, \varphi_{e_3}) \wedge (\varphi_{e_1} \vee \varphi_{e_2} \vee \varphi_{e_3}) \wedge (\varphi_{e_1} \Leftrightarrow \varphi_{e_4}) \wedge (\varphi_{e_2} \Leftrightarrow \varphi_{e_5}) \wedge (\varphi_{e_3} \Leftrightarrow \varphi_{e_6})$$

where $\mathrm{AMO}(x_1, \ldots, x_n)$ is satisfied iff at most one of $x_1, \ldots, x_n$ is satisfied. The formula has three solutions representing the test cases $e_1; e_4, e_2; e_5$ and $e_3; e_6$ which cover the whole specification and avoid the redundancy seen in Example 5.

3.2 Removing Silent Events

It is natural for a system specification to include silent events, e.g. to express that two components in the system synchronize without producing an output observable by the tester. In this case, such a silent event also forms part of $\mathcal{E}$, and the test cases identified by the formula in Section 3.1 are not yet guaranteed to satisfy condition *(i)* of Definition 4. In this section we show how to remove internal events from $\mathcal{E}$ while preserving the causality and conflict relations for the remaining events.

The data structure that we use to represent an event structure does not keep explicit information of the whole causality and conflict relation, but only information about the immediate relations. We associate each event e with the following sets: P_e and S_e consisting of the immediate predecessors and successors of e respectively, and C_e consisting of events e' such that $e \#^i e'$. Algorithm 1 updates the sets P_e and S_e so that the causality relation between the remaining events is preserved (lines 2–5). Also, it propagates the immediate conflict

Algorithm 1 Removal of Silent Events

```
1: for each e in E^τ do
2:     for each p in P_e do
3:         S_p ≔ S_p ∪ S_e \ {e}
4:     for each s in S_e do
5:         P_s ≔ P_s ∪ P_e \ {e}
6:         for each c in C_e do
7:             if NotInConflict(s, c) then
8:                 C_s ≔ C_s ∪ {c}; C_c ≔ C_c ∪ {s}
9:     for each c in C_e do
10:        C_c ≔ C_c \ {e}
11:    E ≔ E \ {e}
```

relation of the silent event to all its immediate successors (line 8). Function NotInConflict is responsible for checking whether two events are already in (not necessarily immediate) conflict.

The non-immediate causality and conflict relations are not stored per se but are computed from the sets S_e, P_e and C_e. For any pair of events $e_i, e_j \in E$, the relation $e_i \leq e_j$ can be computed starting from S_i and recursively traversing its successors until e_j is found, meaning that there is a path from e_i to e_j with arcs in $\bigcup\{ S_k \mid e_i \leq e_k \}$. The relation $e_i \# e_j$ can be computed by checking that there exist events $e_k \in C_l$ and $e_l \in C_k$, i.e. $e_k \#^i e_l$, such that $e_k \leq e_i$ and $e_l \leq e_j$. We show that both relations are preserved after removing the silent events of $\mathcal{E}$.

Proposition 1 *Let $\mathcal{E}$ be an event structure and $\mathcal{E}'$ be the resulting event structure after applying Algorithm 1. For every pair of observable events e_i, e_j, we have $e_i \leq_{\mathcal{E}} e_j$ iff $e_i \leq_{\mathcal{E}'} e_j$.*

Proof. Suppose $e_i \leq_{\mathcal{E}} e_j$, then there exists a path from e_i to e_j in $\mathcal{E}$. Suppose that Algorithm 1 removes an event e in the path. Since line 3 sets $S_p \coloneqq S_p \cup S_e \setminus \{e\}$, the path still exists after removing e: any event reachable from e can be reached from p now. This invariant holds after removing every internal event and therefore the result holds. The counterpart is immediate since causalities are not added, only some immediate ones are removed. □

Proposition 2 *Let $\mathcal{E}$ be an event structure and $\mathcal{E}'$ be the resulting event structure after applying Algorithm 1. For every pair of observable events e_i, e_j, we have $e_i \#_{\mathcal{E}} e_j$ iff $e_i \#_{\mathcal{E}'} e_j$.*

Proof. Whenever an immediate conflict $e \#^i e'$ is removed (while removing event e), for every successor s of e, either s and e' are already in conflict (this is checked by NotInConflict), or the new direct conflict $s \#^i e'$ is added (line 8). Since conflict is inherited w.r.t causality, all the conflicts remain represented. The counterpart is immediate since immediate conflicts are only added whenever the events were not already in conflict. □

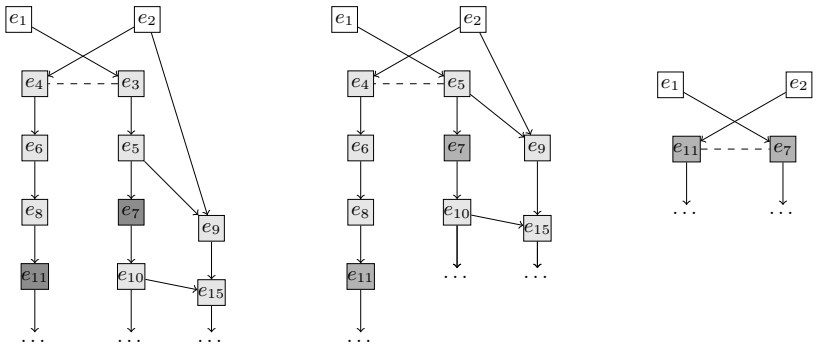


Fig. 5. Removing silent events.

Algorithm 1 does not take in account whether two events are already causally related when it updates S_e and P_e (lines 2–5), potentially leading to redundant (non immediate) relations getting stored. Such redundant relations can be eliminated: we temporarily remove e_i and e_j from P_j and S_i respectively. If e_j is still reachable from e_i, then the information was redundant and can be permanently removed.

Example 7 (Removing Silent Events). Fig. 5 shows how Algorithm 1 works: the structure in the middle is obtained by removing event e_3 from the original prefix while the final result of the algorithm is shown on the right. In the original event structure (left) we have $S_1 = P_5 = C_4 = \{e_3\}$ and $P_3 = \{e_1\}, S_3 = \{e_5\}, C_3 = \{e_4\}$. The structure in the middle is obtained as follows: line 3 updates S_1 to $\{e_5\}$ and line 5 sets $P_5 = \{e_1\}$; since e_5 and e_4 are not in conflict yet, line 8 adds e_4 to C_5 and e_5 to C_4; finally e_3 is removed from C_4 (line 10).

Consider that some events are removed from the prefix in the middle in the following order: e_3, e_5, e_{10}, e_9. Whenever e_5 is removed, its immediate conflict with e_4 is propagated to both e_7 and e_9, however, when e_9 is eliminated, the conflict does not need to be propagated since e_4 and e_{15} are already in (non-immediate) conflict: we have $e_4 \#^i e_7$ and $e_7 \leq e_{15}$. Such situations are handled by the NotInConflict function.

4 Experiments

We implemented a prototype tool called Tours (Testing On Unfolded Reactive Systems) for **co-ioco**-based test-case generation. Tours is based on the Mole unfolding tool [17] with the following main additions:

- variable cut-off criteria, including all-loops coverage by the criterion of [19];
- implementation of the algorithms presented in Section 3;
- computation of the immediate conflict and immediate predecessor relation.

TOURS computes a first over-approximation of $\leq^i$ by inserting all pairs (e, f) such that f consumes a token produced by e. Redundant pairs are then eliminated in the same way as in Section 3.2. The immediate-conflict relation $\#^i$ is obtained by considering all event pairs (e, f) that compete directly for a token, and testing whether there exists a configuration that can be extended with e and with f, but not with both. The latter is a simple variant of a subroutine frequently used by MOLE. TOURS is publicly available under

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

The rest of this section presents experimental results based on two families of examples: a parametric version of the elevator, where we also consider internal behaviors, and an example (called Diamonds) showing how our approach deals with immediate conflict between inputs.

4.1 The Examples

The Elevator Example: we extend the elevator example of Section 2 for several floors and elevators, and we also model its internal behavior. The example is modeled as a network of automata synchronizing on shared actions which can be equivalently captured by a Petri net; we obtain a finite prefix of its unfolding and construct test cases with the SAT encoding using the TOURS prototype.

The system consists of the following components, represented by the automata of Fig. 6 for two floors and one elevator:

Floors: each floor consists of a button that can be pressed to call an elevator. The floor is in an *idle* state where the elevator can be called ($?call_i$), and afterwards 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. Once the elevator is called, it cannot be called again until it returns to the idle state since the $?call_i$ actions are not enabled in the remaining states.

Controllers of elevators: the controller of each elevator e_j starts at an *idle* state and can take a call from any floor f_i. From there 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 at floor f_i, it opens the door ($!open_{j\text{-}i}$) and acknowledges this action to the corresponding floor.

This system is given as an input to the unfolding algorithm (using the all-loops criterion). TOURS returns a prefix whose observable behavior (after removal of silent actions) is shown in Fig. 2 (right). This prefix contains no immediate conflict between inputs, therefore the SAT encoding has a unique solution: the entire prefix. Thus, our method generates exactly one test case in this example.

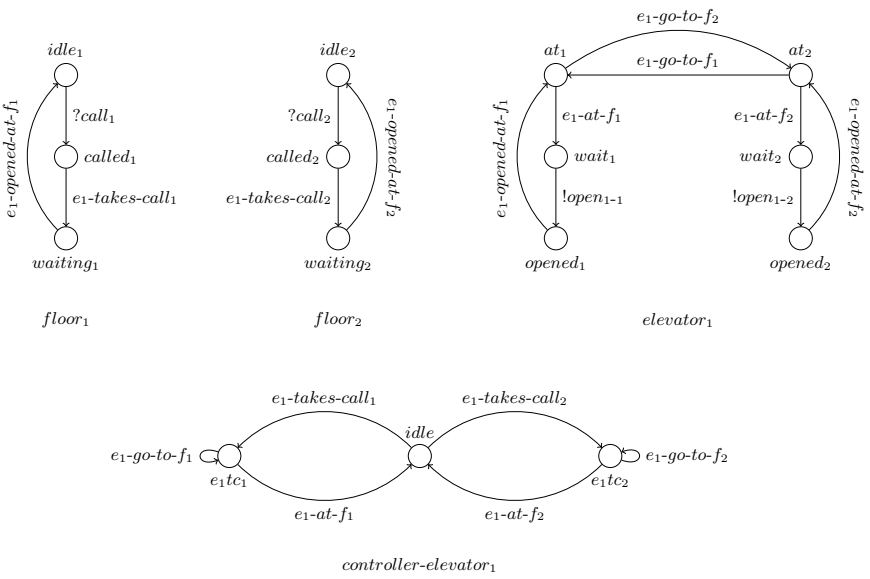


Fig. 6. Network of automata of the elevator example with one elevator and two floors.

Intuitively, the specification of the elevator is that every call at any floor ought to be served eventually (i.e., the door opens at that floor) in any correct implementation; infinite many calls are possible only if infinite many opens happens since the $?call_i$ actions are not allowed at every state. Consider the test case of Fig. 2 (right) and the $?call_2$ action represented by event e_2. This call is followed by an $!open_2$ action in any maximal configuration. Event e_3 corresponds to the scenario where the call is immediately served; e_{10} reflects the fact that the elevator can be called concurrently from another floor ($?call_1$) and that that call can be served first ($e_4 \leq e_{10}$); e_{14} shows that two calls from the first floor (e_1 and e_6) can be served before serving the call from the second floor. The latter shows that there are no priorities between serving different floors; however all the calls are eventually served. A similar analysis can be made for the other call actions.

The example can easily be parametrized to add floors and elevators. If a new floor f_i is added, in addition to adding a new automaton for the floor with transitions e_j-$takes$-$call_i$ 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

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

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

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

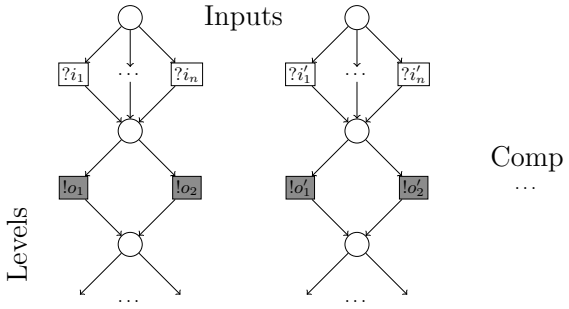


Fig. 7. The diamonds example.

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

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

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

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

The Diamonds Example: We present another example (see Fig. 7) that possesses several global test cases. The example consists on several **components** where the user has a number of choices (**inputs**), after which the system can produce several outputs. This behavior can be repeated several times depending on the **levels** of the components. We run the experiments using different parameters for the components, the inputs and the levels as shown in Table 2.

4.2 The Experimental Setup

In order to make a fair comparison of the algorithms presented in this article and the algorithms of the **ioco** theory, we need to use the same test selection method. Available tools such as TGV [6] or JtorX [22] use test purposes rather than a testing criterion. We therefore proceed as follows to compare with **ioco**: *(i)* the Petri net is translated into its reachability graph; *(ii)* since **co-ioco** coincides with **ioco** in the absence of concurrent events, we apply TOURS to the reachability graph.

In the **ioco** setting, using the all-loops criterion led to test graphs of large size even for simple examples, making it impractical to compute all the test cases. For more meaningful comparisons to be possible, we run the experiments using the original cut-off criterion of MOLE which assures all-transitions and all-states coverage.

4.3 Results

Tables 1 and 2 report the number of events in the unfolding prefix obtained by our method, the number of global test cases (event structures), the number of transitions in an under-approximation[2] of the complete test graph and the number of sequential test cases (labeled transition systems) for the two examples introduced in the last section. The unfolding tool and the SAT encoding consider internal events, while the sizes displayed on the prefix and complete test graph columns only consider observable events.

Floors	Elevators	Prefix	Global Tests	Test Graph	Sequential Tests
2	1	11	1	95	14
2	2	29	1	3929	✗_{SAT}
3	1	43	1	2299	✗_{SAT}
3	2	220	1	3911179	✗_{SAT}
3	3	1231	1	✗_{unf}	✗_{unf}
4	1	219	1	✗_{unf}	✗_{unf}
4	2	1853	1	✗_{unf}	✗_{unf}
4	3	17033	1	✗_{unf}	✗_{unf}

Table 1. The elevator example results.

Comp	Inputs	Levels	Prefix	Global Tests	Test Graph	Sequential Tests
2	1	3	19	1	307	98
2	2	3	37	16	613	794
2	3	3	55	49	919	2938
3	1	1	7	1	133	21
3	1	2	13	1	853	125
3	2	2	25	27	1705	13255

Table 2. The diamonds example results.

We can easily observe the exponential explosion in the number of events when interleavings are used. In addition we see that irrespectively of how many floors or elevators are added in the elevator example, the obtained global test case is always unique since the example does not introduce conflict between input events. In contrast, the number of sequential test cases increases in the interleaving setting since concurrency is transformed into conflict. The diamonds example introduces conflicts between inputs generating several global test cases, however the number of tests can still be exponentially smaller than in the sequential case.

The ✗_{unf} symbol indicates that the unfolding tool was not able to obtain a finite prefix (complete test graph), while the ✗_{SAT} symbol indicates that the

[2] This graph is not output-complete.

SAT solver was not able to find solutions (for more than 3 floors and 2 elevators, we were not able to run the SAT solver with interleaving semantics since the unfolding had not finished).

The unfolding of the Petri 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. Since causality is transitive and conflict 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 as explained in Section 3.2 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.

5 Conclusion

This paper shows the advantages of using true-concurrency models to describe the behavior of test cases in concurrent systems. We have shown how to split a finite prefix of the specification's unfolding into a test suite even in the presence of internal actions. Finally, the results of this article have been implemented in the prototype tool Tours and run on several examples showing the advantages of our method compared with traditional **ioco** test-case generation algorithms.

The obtained global test cases are not meant to be actually executed, 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, further machinery is needed [21]. We will study the concretization of the generated abstract test cases into inputs that can be given to the actual system under test to allow the automatic execution of test cases and thus completely automate the testing procedure. A possible approach is to consider labeling actions as a symbolic representation of the input and output domain and apply SMT for the concretization.

Future work also includes a tighter integration of the test-case generation part of Tours with the unfolding component to improve its performance.

References

1. Tretmans, J.: Model based testing with labelled transition systems. In: Formal Methods and Testing. Volume 4949 of LNCS. (2008) 1–38
2. Heerink, L., Tretmans, J.: Refusal testing for classes of transition systems with inputs and outputs. In: Proc. FORTE. IFIP 107 (1997) 23–38
3. Jéron, T.: Symbolic model-based test selection. Electronic Notes in Theoretical Computer Science **240** (2009) 167–184
4. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design **34**(3) (2009) 238–304
5. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations for the distributed test architecture. In: Proc. FATES. Volume 5047 of LNCS. (2008) 200–215

6. Jard, C., Jéron, T.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer **7**(4) (2005) 297–315
7. Hennessy, M.: Algebraic theory of processes. MIT Press series in the foundations of computing. MIT Press (1988)
8. Peleska, J., Siegel, M.: From testing theory to test driver implementation. In: Formal Methods Europe. Volume 1051 of LNCS. (1996) 538–556
9. Schneider, S.: Concurrent and Real Time Systems: The CSP Approach. 1st edn. John Wiley & Sons, Inc., New York, NY, USA (1999)
10. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. Formal Methods in System Design **20**(3) (2002) 285–310
11. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - A survey. In: Proceedings of the IEEE. Volume 84. (1996) 1090–1123
12. Jourdan, G., von Bochmann, G.: On testing 1-safe petri nets. In: TASE 2009, Third IEEE International Symposium on Theoretical Aspects of Software Engineering, 29-31 July 2009, Tianjin, China. (2009) 275–281
13. von Bochmann, G., Jourdan, G.: Testing k-safe petri nets. In: Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, 2009. Proceedings. (2009) 33–48
14. Jard, C.: Synthesis of distributed testers from true-concurrency models of reactive systems. Information & Software Technology **45**(12) (2003) 805–814
15. Ulrich, A., König, H.: Specification-based testing of concurrent systems. In: Proc. FORTE. Volume 107 of IFIP Conference Proceedings. (1997) 7–22
16. Ponce de León, H., Haar, S., Longuet, D.: Model-based testing for concurrent systems with labeled event structures. STVR **24**(7) (2014) 558–590
17. Schwoon, S.: The MOLE unfolding tool. `http://www.lsv.ens-cachan.fr/~schwoon/tools/mole/`
18. Nielsen, M., Plotkin, G.D., Winskel, G.: Petri nets, event structures and domains, part I. Theoretical Computer Science **13** (1981) 85–108
19. Ponce de León, H., Haar, S., Longuet, D.: Unfolding-based test selection for concurrent conformance. In: Proc. ICTSS. Volume 8254 of LNCS. (2013) 98–113
20. Jéron, T., Morel, P.: Test generation derived from model-checking. In: Proc. CAV. Volume 1633 of LNCS. (1999) 108–121
21. Ponce de León, H., Haar, S., Longuet, D.: 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., Springer (2014) 369–387
22. Belinfante, A.: JTorX: A tool for on-line model-driven test derivation and execution. In: Proc. TACAS. Volume 6015 of LNCS. (2010) 266–270
23. Henniger, O.: On test case generation from asynchronously communicating state machines. In: International Workshop on Testing Communicating Systems. IFIP Conference Proceedings. Springer (1997) 255–271
24. Kähkönen, K., Saarikivi, O., Heljanko, K.: Using unfoldings in automated testing of multithreaded programs. In: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012. (2012) 150–159