Symbolic Observation Graph-Based Generation of Test Paths

Kais Klai, Mohamed Taha Bennani, Jaime Arias, Jörg Desel, Hanen Ochi

Symbolic Observation Graph-Based Generation of Test Paths

Kais Klai[1], Mohamed Taha Bennani[2], Jaime Arias[1]✉, Jörg Desel[3], and Hanen Ochi[4]

[1] Université Sorbonne Paris Nord, LIPN UMR CNRS 7030, Villetaneuse, France
{kais.klai, jaime.arias}@lipn.univ-paris13.fr
[2] University of Tunis El Manar, Faculty of Sciences of Tunis, Tunis, Tunisia
taha.bennani@fst.utm.tn
[3] Fern Universität in Hagen, Hagen, Germany
joerg.desel@fernuni-hagen.de
[4] EFREI Paris, EFREI Research Lab,Villejuif, France
hanen.ochi@efrei.fr

Abstract. The paper introduces a theoretical foundation for generating abstract test paths related to Petri net specifications. Based on the structure of the Petri net model of the system, we first define the notion of *unobservable* transition. Unless such a transition is unreachable, we prove that its firing is necessarily ensured by the firing of another transition (namely *observable* transition) of the Petri net. We show that the set of *observable* transitions is the smallest set that guarantees the *coverage* of all the transitions of the Petri net model, i.e., any set of firing sequences of the model, namely *observable traces*, involving all the *observable* transitions passes eventually through the *unobservable* transitions as well. If some unobservable transitions are mandatory to trigger the execution of a test sub-sequence, observable traces are completed with such transitions to enhance the controllability of the test scenario. In addition to structurally identifying observable (and unobservable) transitions, we mainly propose two algorithms: the first allows to generate a set of observable paths ensuring full coverage of all the system transitions. It is based on an on-the-fly construction of a hybrid graph called the *symbolic observation graph*. The second algorithm completes the observable paths in order to explicitly cover the whole set of system's transitions. The approach is implemented within an available prototype, and the preliminary experiments are promising.

Keywords: Model-based testing, transition coverage, Petri nets.

1 Introduction

Model-based testing automates a set of processes, namely the generation of test cases from models, the derivation of executable scripts, and the execution of test cases or test scripts [45]. The cornerstone of model-based testing is the generation of test inputs from the behavioural model of the system under test. Several techniques have been introduced for test inputs generation, such as solvers

[34], constrained logic programming [32], search-based algorithms [38] and model checking [19]. Except for some educational systems, the number of input data is often infinite. Therefore, testing cannot be exhaustive. Thus, these techniques seek to reduce the number of test inputs by modelling sets of similar behaviours through coverage criteria. As a result, an abstract test path would characterize a sequence of actions or events a system must perform to achieve a certain behaviour, represented by a set of requirements of a coverage criterion.

Test inputs generation satisfying structural or behaviour coverage criteria suffers from the well known state space explosion problem [41]. Several approaches have offered solutions to cope with this problem such as symmetry reduction [17], live variable reduction [18], cone of influence [13], slicing [11], transition merging [50], partial order reduction [42], τ-confluence [33], and simultaneous reachability analysis [44]. *Symbolic Observation Graph* (SOG) [20,25] represents an efficient technique to reduce the state space graph based on the observation of the pertinent atomic propositions of a temporal formula to be verified. In this paper, we adopt the SOG technique for the input generation perspective.

Given a finite system modelled with a Petri net [37], our goal is to generate a set of firing sequences that ensure the coverage of the whole set of the model transitions. A Petri net structure-based solution to identify the set of significant transitions, which we call *observable*, is first proposed. We show that the choice of the test inputs that allow the system behaviour to cover this set of transitions guarantees the coverage of all the model transitions of the system to be tested. *Unobservable* transitions are then guaranteed to be covered by observable transitions unless they belong to an unobservable cycle in the model (i.e., cycles with unobservable transitions only), in which case the transitions of such a cycle are proven to be dead. Such cycles can be detected and removed by a simple browsing of the Petri net model. This discrimination allows us to build, on-the-fly, the SOG w.r.t. the observable transitions, reducing considerably the state space to be traversed to cover all the transitions of the system. During the construction of the SOG, we aim to find paths that cover all the observable transitions. As soon as this goal is reached, the construction of the SOG is stopped. If necessary, the obtained paths can be completed by backtracking with unobservable transitions to obtain *complete abstract* sequences involving all the transitions of the system.

The rest of this paper is organized as follows: in Section 2, required concepts and formalisms are presented. Section 3 and Section 4 are the core of the paper, where the proposed approach is presented. In the former, we show how to use the structure of the Petri net model to partition the transitions into two subsets: observable and unobservable transitions, and we establish underlying theoretical results. In the latter, we show how to check the coverage of all Petri net model's transitions by exploiting the SOG. The different algorithms were implemented in a software tool, and the obtained results of our experiments are reported in Section 5. Before concluding and presenting directions for future work in Section 7, we discuss related work and give their pros and cons in Section 6.

2 Preliminaries

2.1 Labelled Transition Systems (LTS)

Definition 1 (Labelled Transition Systems). *A* labelled transition system *(LTS for short) $\mathcal{T}$ is a 4-tuple $(S, Act, \rightarrow, s_0)$ where:*

- *S is a (finite) set of states;*
- *Act is a (finite) set of actions;*
- *$\rightarrow \in S \times Act \times S$ is a transition relation; and*
- *s_0 is the initial state.*

The set of actions of an *LTS* could be divided into two disjoint subsets of observable (namely *Obs*) and unobservable transitions (namely *UnObs*). Determining which transitions are observable, and which ones are not, depends on the approach/objective/domain. In our approach, the precise (unique) definition of unobservable/observable transitions is presented in Section 3. For instance, Fig. 1 shows an *LTS* where transitions t_7, t_8, t_{10} and t_{11} (coloured) are observable.

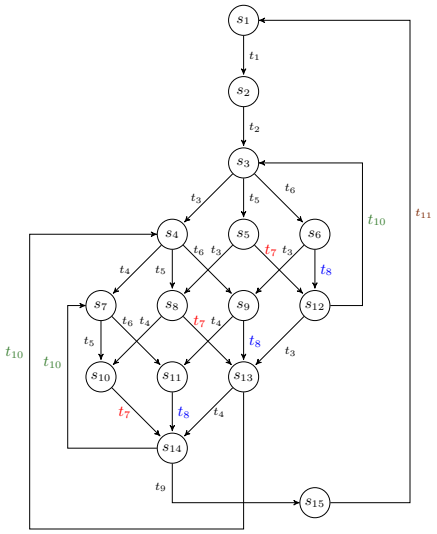

Fig. 1. An *LTS*

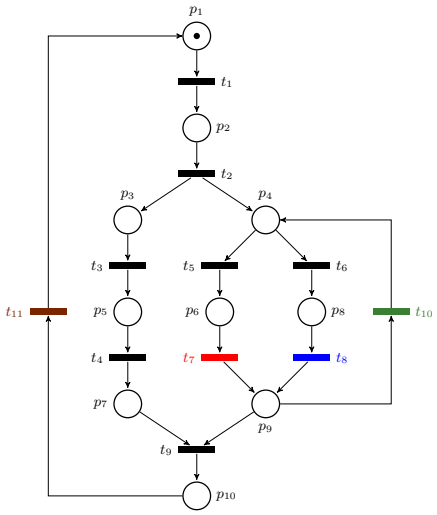

Fig. 2. A Petri net

2.2 Petri Nets

Definition 2 (Syntax). *A* Petri net *is a tuple $\mathcal{N} = (P, T, F, W)$, where:*

- *P is a finite set of places;*
- *T is a finite set of transitions;*

- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (as one-way arrows) connecting places to transitions and transitions to places; and*
- $W : (P \times T) \cup (T \times P) \to \mathbb{N}$ *represents arcs' weight.*

For the sake of simplicity and without loss of generality, we assume, in this paper, that the weight of the edges are all equal to 1. A marking of a Petri net $\mathcal{N}$ is a function $m : P \to \mathbb{N}$ assigning a number of tokens to each place. A Petri net model is generally associated with an initial marking, denoted by m_0, that represents the initial state of the underlying system. Fig. 2 illustrates a Petri net example where the places are represented by circles and transitions by rectangles. The initial marking is such that place p_1 contains one token and all the other places are empty. Each node $x \in P \cup T$ of the net has a pre-set and a post-set defined respectively as follows: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$, and $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. Adjacent nodes are then denoted by ${}^\bullet x^\bullet = {}^\bullet x \cup x^\bullet$.

Semantics and Notation A transition t is said to be enabled by a marking m (denoted by $m \xrightarrow{t}$) iff $\forall p \in {}^\bullet t,\ W(p,t) \leq m(p)$. If a transition t is enabled by a marking m, then its firing leads to a new marking m' (denoted by $m \xrightarrow{t} m'$) s.t. $\forall p \in P : m'(p) = m(p) - W(p,t) + W(t,p)$. For a finite sequence of transitions $\sigma = t_1 \dots t_n \in T^*$, $m_0 \xrightarrow{\sigma}$ denotes the fact that σ is enabled by m_0, i.e., $\exists m_0, \dots m_n$ s.t., $m_0 \xrightarrow{t_1} m_1 \xrightarrow{t_2} \dots \xrightarrow{t_n} m_n$. $m \xrightarrow{\sigma} m'$ denotes the fact that the firing of σ from marking m leads to marking m'. $\bar{\sigma}$ denotes the set of transitions occurring in σ.

The language of finite firing sequences associated with a marked Petri net $(\mathcal{N}, m_0)$ is then defined as $L((\mathcal{N}, m_0)) = \{\sigma \in T^* \mid m_0 \xrightarrow{\sigma}\}$. Given a set of markings S, we denote by $Enable(S)$ the set of transitions enabled by elements of S. The set of markings reachable from a marking m in $\mathcal{N}$ is denoted by $R(\mathcal{N}, m)$. A transition t is said to be a *dead* transition in $(\mathcal{N}, m_0)$ if it is enabled in no marking in $R(\mathcal{N}, m_0)$. The set of markings reachable from a marking m, by firing transitions of a subset T' only, is denoted by $Sat(m, T')$. By extension, given a set of markings S and a set of transitions T', $Sat(S, T') = \bigcup_{m \in S} Sat(m, T')$. The graph of reachable markings of a marked Petri net $(\mathcal{N}, m_0)$, denoted $G(\mathcal{N}, m_0)$, is an *LTS* whose nodes S are the set of markings $R(\mathcal{N}, m_0)$, and the arcs are labelled with the transitions of $\mathcal{N}$. The initial node is the initial marking m_0, and a node (marking) m' is the successor of a node m iff $\exists t \in T$ s.t. $m \xrightarrow{t} m'$. The previous notations on markings of a Petri net are extended (and used in the rest of the paper) for states of an *LTS*. The *LTS* of Fig. 1 represents the reachability graph corresponding to the Petri net of Fig. 2.

2.3 Symbolic Observation Graph (SOG)

The SOG induced by a given LTS with transitions partitioned into observable and unobservable ones is defined as an *LTS* where nodes, called *aggregates*, are sets of single states connected by unobservable transitions, and compactly encoded by decision diagram techniques (e.g., BDDs [22]). The edges of the SOG are however labelled with observable transitions only.

Definition 3 (Aggregate). *Let $\mathcal{T} = (S, Act, \rightarrow, s_0)$ be an LTS, where $Act = Obs \cup UnObs$ (with $Obs \cap UnObs = \emptyset$). An* aggregate *$a$ is a non-empty subset of S satisfying $s \in a \Leftrightarrow Sat(s, UnObs) \subseteq a$.*

Definition 4 (Symbolic Observation Graph (SOG)). *A* symbolic observation graph *associated with an LTS $\mathcal{T} = (S, Act, \rightarrow, s_0)$ is an LTS $\mathcal{G} = (S', Act', \rightarrow', a_0)$ such that:*

- *S' is a finite set of aggregates satisfying:*
  - *$\star$ $\forall a \in S'$, $\forall t \in Act'$, $\exists (s, s') \in a \times S$: $s \xrightarrow{t} s' \Leftrightarrow \exists a' \in S' : a' = Sat(\{s' \in S \mid \exists s \in a \wedge s \xrightarrow{t} s'\}, UnObs) \wedge (a, t, a') \in \rightarrow'$;*
- *$Act' = Obs$;*
- *$\rightarrow' \subseteq S' \times Act' \times S'$ is the transition relation, obtained by applying $\star$; and*
- *a_0 is the initial aggregate s.t. $a_0 = Sat(s_0, UnObs)$.*

Point $\star$ of the previous definition deserves explanation. By Definition 3, an aggregate contains the maximal set (fix-point computation with *Sat* function) of states linked by unobservable transitions. An arc (a, t', a'), connecting an aggregate a to an aggregate a', labelled with an observable transition t', must exist in the SOG iff the set of states in a enabling t' is not empty, and any state that is reachable by any sequence $t'.\sigma$, where $\sigma \in UnObs^*$, is necessarily in a'.

The SOG construction algorithm is presented in [20]. Despite the exponential theoretical complexity of its construction (a single state could belong to different aggregates), the SOG has in practice a rather moderate size comparing to the explicit representation of the corresponding *LTS* (see [20,26,25] for experimental results). Fig. 3 shows the SOG related to the *LTS* example of Fig. 1 based on the observation of transitions t_7, t_8, t_{10} and t_{11}. Notice that, in this figure, the aggregate internal sub-graphs as well as the connection between explicit states belonging to different aggregates, are showed for the readability purpose. The internal set of states is represented by a binary decision diagram (BDD) [22], while for observable transition labels, only one solid edge connects two aggregates.

2.4 Test Coverage Criteria

Since the evaluation of all system's inputs values is not possible, testers must rely on measurements' features to argue the trust they can place in the system under test. Coverage criteria define test objectives or requirements that test entries (deduced from test cases) strive to satisfy. Therefore, if the test inputs meet the test objectives, the confidence placed by a tester in the system will match the exigency of the coverage criterion.

Several coverage criteria related to Petri nets are defined in [40], e.g., Structural and Behavioural Analysis Coverage (SBAC), and Concurrent Session Behaviour Coverage Criteria (CSBCC). This paper deals with the transition coverage criterion. That is, a test suite (i.e., a set of firing sequences) that covers this criterion must fire any transition of a Petri net at least once.

Definition 5 (Transition Coverage). *Let $\mathcal{N} = (P, T, F, W)$ be a Petri net, and $\mathcal{T}_s = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ be a test suite. $\mathcal{T}_s$ satisfies* transition coverage *criterion iff $\forall t \in T$, $\exists \sigma \in \mathcal{T}_s$, $t \in \bar{\sigma}$.*

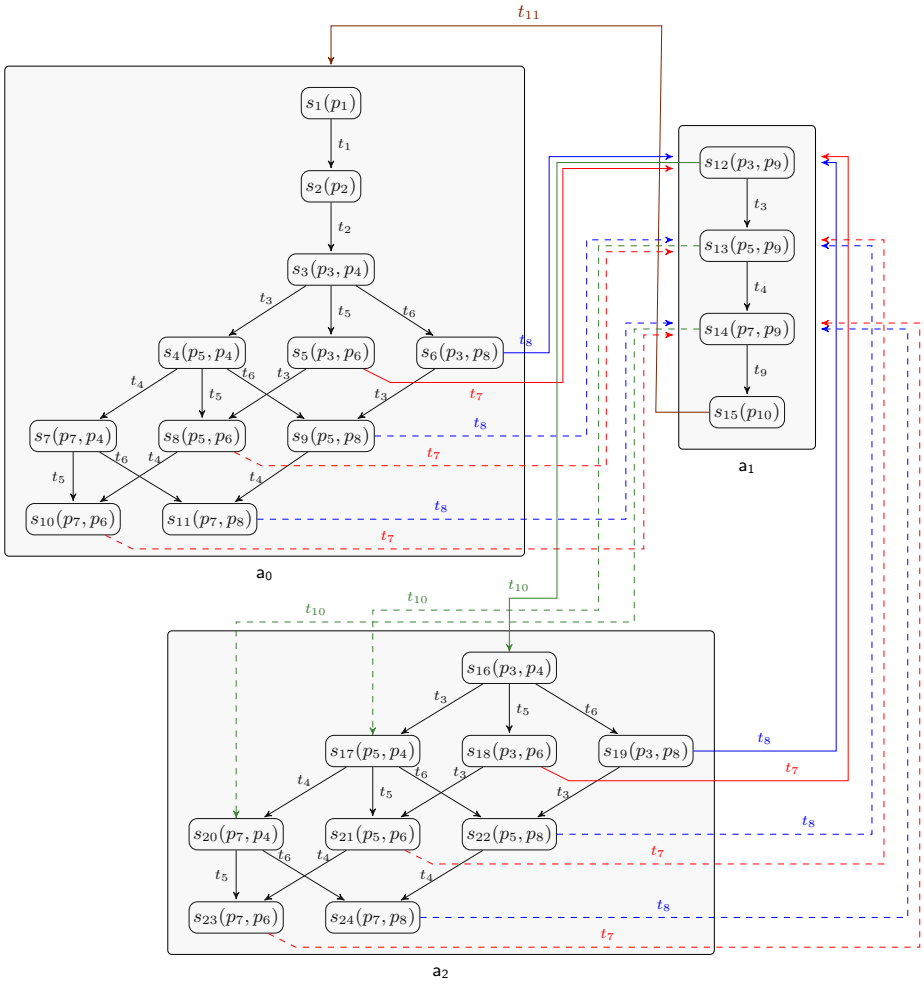

Fig. 3. SOG related to the *LTS* of Fig. 1 under the observation of $\{t_7, t_8, t_{10}, t_{11}\}$

3 Structure-Based Coverage Relation for Petri Nets

In this section, we illustrate how covering a reduced set of transitions can lead to the coverage of all the system's transitions. To reach this goal, we propose an original approach based on the identification of a specific structural pattern in a Petri net model. Such a pattern allows to deduce behavioural information related to the firing of the transitions of the model.

Definition 6. *The coverage relation, denoted by* $\triangleright$ *is the transitive relation defined as follows:* $\triangleright : T \rightarrow T$ *s.t.* $\forall t' \in T,\ t' \triangleright t$ *implies that any firing sequence containing* t' *contains necessarily* t*.*

In the following, we define *unobservable* transitions.

Definition 7. *A transition t in a marked Petri net $(\mathcal{N}, m_0)$ is said to be* ***unobservable*** $\iff \exists\ p \in t^\bullet,\ {}^\bullet p = \{t\} \wedge p^\bullet \neq \emptyset \wedge m_0(p) = 0$. *For a given transition t, the set of such places is denoted by $t^\sim$.*

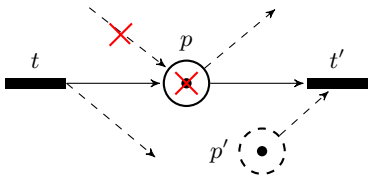

Fig. 4. Unobservable transition t

Informally (see Fig. 4), an unobservable transition t is a transition having, at least, one unmarked output place p such that no other transition can produce tokens in p (${}^\bullet p = \{t\}$). Thus, p being initially unmarked, the firing of any output transition of p is impossible before firing t. For this reason, any firing sequence containing any transition $t' \in p^\bullet$ contains necessarily t (i.e., $t' \triangleright t$).

Example 1. Consider the marked Petri net of Fig. 2. The set of unobservable transitions is $\{t_1, t_2, t_3, t_4, t_5, t_6, t_9\}$. Transitions t_7, t_8, t_{10} and t_{11} are not. Indeed, each of the transitions t_7, t_8 and t_{10} has a single output place that has another input transition, while t_{11} is not unobservable since place p_1 is initially marked.

In the rest of the paper, each transition that is not *unobservable* is said to be *observable*. Given a marked Petri net $(\mathcal{N}, m_0)$, the set T of its transitions can then be divided into the two disjoint subsets *Obs* and *UnObs* that represent the observable and the unobservable transitions, respectively.

Next, we establish some theoretical results about the fireability of a transition depending on its type (observable or unobservable). The proof is in Appendix A.

Lemma 1. *Given a marked Petri net $(\mathcal{N}, m_0)$ and a transition t, $t \in UnObs \implies \forall p \in t^\sim,\ \forall t' \in p^\bullet,\ t' \triangleright t$.*

Corollary 1. *Given a marked Petri net $(\mathcal{N}, m_0)$ and a transition $t \in UnObs$. Then, t is dead $\implies \forall p \in t^\sim,\ \forall t' \in p^\bullet,\ t'$ is dead.*

The proof of the previous corollary is immediate from Lemma 1. In fact, if an unobservable transition t is dead (i.e., there is no firing sequence $\sigma.t$ from the initial marking m_0), none of the output places in $t^\sim$ will be marked, preventing to fire their output transitions. In the following lemma, an *unobservable cycle* denotes a cycle in the Petri net model that contains unobserved transitions only. The proof is presented in Appendix B.

Lemma 2. *Let $(\mathcal{N}, m_0)$ be a marked Petri net. Then, $\forall t \in UnObs$, $\exists t' \in Obs$: $t' \triangleright t$, or t belongs to an unobservable cycle.*

Our ultimate goal is to generate a set of traces that cover all the transitions of the system (represented by a Petri net). Instead of considering all the transitions of the system, we consider only the observable transitions (Obs) and look for sequences that cover these transitions. In general, covering observable transitions implies the covering of the unobservable ones. However, as established in the previous lemma, this does not hold when the Petri net model contains unobservable cycles. In this case, all the transitions of an unobservable cycle are dead. The proof is presented in Appendix C.

Lemma 3. *Given a marked Petri net $(\mathcal{N}, m_0)$, all the transitions belonging to an unobservable cycle are dead.*

Next, we present the main result of our approach: if there exists a set of firing sequences $\sigma_1 \dots \sigma_n$ covering Obs, then all the transitions of the Petri net are covered by these sequences except those belonging to unobservable cycles.

Theorem 1. *Let $(\mathcal{N}, m_0)$ be a marked Petri net. Let Obs be the set of observable transitions (i.e. any transition not satisfying Definition 7). If $\exists \sigma_1 \dots \sigma_n \in L((\mathcal{N}, m_0))$ s.t. $Obs \subseteq \bigcup_{i=1}^{n} \bar{\sigma_i}$, then $\forall t \in T$ one of the two following holds:*

1. *$\exists i \in \{1 \dots n\}$ s.t. $t \in \bar{\sigma_i}$.*
2. *t belongs to an unobservable cycle.*

Proof. The proof is trivial using Lemma 2.

In the following theorem, we state that the set of observable transitions Obs is the smallest set ensuring the coverage of all the transitions of a Petri net (except those belonging to an unobservable cycle whose transitions are necessarily dead). The proof is presented in Appendix D.

Theorem 2. *$\forall Obs' \subseteq Obs$, Obs' satisfies Theorem 1 $\Longrightarrow$ $Obs' = Obs$.*

Limit cases: It is worth noting that there are two limit cases to our approach:

1. $Obs = \emptyset$. This happens when the whole set of transitions of the Petri net is involved in a dead cycle.
2. $Obs = T$ (i.e., all the transitions are observable). This happens, for instance, when none of the places has one input transition only. Such a case is obviously not favourable to our approach.

4 Test Specification Computation

The main theoretical result of the previous section is that covering the structurally computed observable transitions ensures the coverability of the whole set of system's transitions. It is then sufficient to find a collection of firing sequences

with all observable transitions to cover all system's transitions. In this section, we propose a SOG-based approach to generate traces of the system that pass through the observable transitions. Such traces are first generated as the projection of full traces of the system on observable transitions only (called *observable traces/paths*). The construction of the SOG is revisited in our work in order to generate these observable traces on-the-fly (i.e., the construction is stopped as soon as all the observable transitions are covered). Once the observable traces are generated, the full traces (called *abstract paths*) are generated using a symbolic algorithm (BDD-based operations) based on a backward traversal of the SOG's nodes involved by these observable traces. Without loss of generality, we assume in this section that the system has no unobservable cycles. Indeed, one can detect (and remove) them using a structural analysis of the Petri net model.

4.1 On-The-Fly SOG-Based Generation of Observable Traces

The purpose of Algorithm 1 is to extract the observable traces during the construction of the SOG. Given the system model and the set of observable transitions, the goal of the algorithm is to collect a test suite $\mathcal{T}_s = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ such that, for any observable transition o, there exists an observable trace σ_i containing an occurrence of o.

Algorithm 1 is based on a depth-first search (DFS) traversal during the construction of the SOG. It takes as input the Petri net $\mathcal{N}$, its initial marking m_0, and the set of observable transitions Obs. The main used data structures are: (1) a stack "st" containing the aggregates "a" not yet completely treated, associated with the not yet treated observable transitions fireable from "a" and computed with the function `EnableObs`; (2) a set of sequences $\mathcal{T}_s$ to be calculated (the output of the algorithm); (3) the current trace $curT$; and (4) the set of covered observable transitions Cov. The initialization step of the algorithm (lines 4-6) computes the SOG's first aggregate using function `InitialAggregate`. It also initializes stack "st" with the first aggregate, and its fireable observable transitions are identified with function `EnableObs`. An iteration of the main loop (lines 7-28) picks and processes an item (a, f) from the stack. The algorithm ends in two cases: (1) the set of observable transitions Obs is fully covered, and (2) the stack is empty i.e., the SOG is completely built without covering the dead observable transitions. At each step of the main loop, we pick up the top couple (a, f). If the set of enabled transitions is not empty (line 9), then we choose (and remove from f) a transition to fire using the function `chooseTransition` that favours transitions that have not been covered yet, if any. Such a function could be more sophisticated to rely on some objective function (e.g., the length and/or the number of generated observable traces). If the selected transition o is the last observable transition to be covered, we save the current trace in $\mathcal{T}_s$ before leaving the loop (lines 13-15). Then, we push back the couple (a, f) to the stack and compute the successor a' of a by transition o (lines 16-17). If the successor already exists (lines 18-20), we add the current trace to $\mathcal{T}_s$ and then remove the last transition from the current path. In fact, we just finished traversing a path leading to a' where the last transition is o (such a trace has to be saved in $\mathcal{T}_s$),

Algorithm 1: Generation_of_observable_traces

```
   Data: N, m_0, Obs
   Result: T_s
 1 Stack<aggregate, set<transition>> st; Set of traces T_s; aggregate a, a';
 2 Current observable trace curT; Set of covered observable transitions Cov;
 3 begin
 4 |  Cov ← ∅;
 5 |  a ← InitialAggregate((N, m_0)) ;
 6 |  st.push(a, EnableObs(a));
 7 |  while (st ≠ ∅ ∧ Cov ⊂ Obs) do
 8 |  |  st.pop(a,f);
 9 |  |  if (f ≠ ∅) then
10 |  |  |  o ← chooseTransition(f, Cov);
11 |  |  |  Cov = Cov ∪ {o};
12 |  |  |  curT ← curT.o;
13 |  |  |  if (Cov == Obs) then
14 |  |  |  |  T_s ← T_s ∪ {curT}
15 |  |  |  end
16 |  |  |  st.push(a,f);
17 |  |  |  a' ← Succ(a, o);
18 |  |  |  if (a' already exist) then
19 |  |  |  |  T_s ← T_s ∪ {curT}
20 |  |  |  |  curT ← curT − curT.last;
21 |  |  |  else
22 |  |  |  |  st.push(a', EnableObs(a'));
23 |  |  |  end
24 |  |  else
25 |  |  |  T_s ← T_s ∪ {curT}
26 |  |  |  curT ← curT − curT.last;
27 |  |  end
28 |  end
29 end
```

and we are going in the next iteration to come back to aggregate a to explore another path (o must be removed from the current trace). In case the aggregate a' is a new aggregate, we push it on the stack with the corresponding set of enabled observable transitions (line 22). Finally, in case the picked aggregate is completely treated (lines 24-26), we add the current trace to $\mathcal{T}_s$ and then remove the last transition from the current path. The reason is the same as when the compute successor aggregate exists. Notice that the current trace $curT$ is added to $\mathcal{T}_s$ only if there exists at least one transition in $curT$ that is newly covered.

Example 2. When Algorithm 1 gets the Petri net of Fig. 2 as a first parameter $\mathcal{N}$, an initial marking m_0 where only p_1 contains a token, and a set of observed transitions $Obs = \{t_7, t_8, t_{10}, t_{11}\}$, the initialization operation (line 5) generates the aggregate a_0 of Fig. 3. As there are two outgoing transitions, t_7 and t_8,

identified by `EnableObs` (line 6), when the algorithm picks one of them in line 10, the statement of line 17 will create a_1. Aggregate a_1 has two outgoing transitions: t_{10} and t_{11}, where the former conducts to a_2. The last created aggregate leads to a previously created one a_1 through t_7 or t_8. At this step, line 19 inserts the variable $curT$ containing $< t_8, t_{10}, t_7 >$ into the set of observed traces identified by the variable $\mathcal{T}_s$. Using the backtracking mechanism implemented in lines 20 and 26, the algorithm goes back to the aggregate a_1 to explore the last uncovered transition t_{11} leading to a previously explored aggregate a_0. Therefore, line 19 generates a new observed trace $< t_8, t_{11} >$. In this final iteration, the stack st is empty, and all transitions are covered.

4.2 Extracting Abstract Traces

The set of observable traces could be important to trigger particular executions of the system. However, in case we need to have complete executions involving the totality of the system transitions, we can extract such traces from the observable ones. Therefore, we will insert a sequence of unobserved transitions between two observed ones generated by the previous algorithm, starting from the initial marking state. Several criteria could be used to choose the unobserved sequence, such as test feasibility, balanced combinations, and sequence length. In our case, we aim to minimize the sequence's length traversing an aggregate.

Algorithm 2: Generation_of_abstract_path

```
   Data: obsPath, SOG, UnObs
   Result: abs_path
 1 sequence<transition> abs_path, path_agr; aggregate agr; Set exitpts source;
 2 transition trans; Pair<aggregate, Set entrypts> arg_entry;
 3 Stack<pair<aggregate, Set entrypts>> entry_points; Set entrypts target;
 4 begin
 5     trans=obsPath.pop();
 6     entry_points = research_entry_points(obsPath, SOG);
 7     abs_path=Null ;                    /*entry marking of next aggregate*/
 8     while (entry_points not empty) do
 9         agr_entry = entry_points.pop();
10         target = agr_entry.second;
11         agr = agr_entry.first;
12         source = FirableObs(agr, trans);            /*exit points of agr*/
13         path_agr = sub_path_aggregate(source, target, agr, UnObs);
14         abs_path = path_agr + trans + abs_path;
15         trans = obsPath.pop();
16     end
17 end
```

Algorithm 2 takes as input the observable path `obsPath`, the SOG built earlier `SOG`, and the set of unobservable transitions `UnObs`, and it returns the ab-

stract path `abs_path`. During the first phase (lines 5-7), the algorithm extracts the observed path's last transition, identifies the aggregates' entry points set, and initializes the abstract path. The function `research_entry_points` calculates and stores in the stack (*entry_points*) the list of aggregates traversed along the observable path `obsPath`, each associated with the set of its input states (i.e., the set of states reached by the previous observable transition in the trace starting from the previous aggregate). As long as there are items in the stack (aggregates and their entry points), the second phase (lines 8-16) uses a backtracking traversal of the current aggregate (at the top of the stack) in order to build a trace of unobservable transitions linking the *source* set of states (i.e., the ending points in an aggregate that will lead to the next aggregate while traversing the path) to the *target* set of states (i.e. entry points of the same aggregate). The above process is done by the function `sub_path_aggregate`. The last function uses a BFS strategy to provide the shortest unboservable path linking the entry and exit points of the aggregate.

Example 3. To apply Algorithm 2 to our illustrative example, the first parameter *obsPath* contains one of the two observed paths provided by the previous algorithm: $< t_8, t_{10}, t_7 >$ and $< t_8, t_{11} >$. If the second parameter is described by Fig. 3, the third one, called *UnObs*, contains all transitions of the Petri net of Fig. 2, except the observed transitions t_7, t_8, t_{10}, and t_{11}. For the first unobserved path $< t_8, t_{10}, t_7 >$, the variable *trans* stores t_7, and line 6 computes the entry points of the aggregates a_0, a_1, and a_2, that is, $\{s_1\}$, $\{s_{12}, s_{13}, s_{14}\}$, and $\{s_{16}, s_{17}, s_{20}\}$, respectively. The *entry_points* stack holds three couples, where $(a_2, \{s_{16}, s_{17}, s_{20}\})$ is on the top. After removing the pair from the stack, lines (10-11) isolate the set of entry points and the aggregate. The former is stored in the *target* variable, and the latter in the *agr* variable. Then, line 12 computes the states' set in the aggregate a_2 that can fire the transition t_7. For our example, *source* variable contains $\{s_{18}, s_{21}, s_{23}\}$. Before extracting the next unobserved transition t_{10} and proceeding to the next iteration, line 13 identifies the path from the entry point of the aggregate a_2 to the observed transition t_7. The generated path is extended, in line 14, by adding the transition t_7 and an empty abstract path, as we are at the first iteration. In this case, we have three different alternatives: $< t_5, t_7 >$, $< t_3, t_5, t_7 >$, or $< t_3, t_4, t_5, t_7 >$. Our function `sub_path_aggregate` chooses the first state that can fire t_7. The actual implementation generates the first abstract path. After two iterations successively processing t_{10} and t_8 and the aggregates a_1 and a_0, respectively, the algorithm returns the abstract path $< t_1, t_2, t_6, t_8, t_{10}, t_5, t_7 >$.

5 Experiments

In this section, we compare the performance of our approach with that of two state-of-the-art tools: `MISTA`[5] and `NModel`[6]. This choice is based on the tools

[5] https://github.com/dianxiangxu/MISTA
[6] https://github.com/juhan/NModel

presented in the reviews [43,8,35,46]. We selected those that are (1) open-source, (2) available and maintained, (3) take as input either an FMS or a Petri net, (4) support the transition coverage criteria, and (5) generate all the paths.

Our approach has been implemented in the open-source tool `sogMBT` [3], that is written in C++ and has been integrated to the user-friendly web platform `CosyVerif`[7]. A total of 8 models from the Model Checking Contest[8] were used in our experiments: *Philosophers* (`philo`), *Referendum* (`referendum`), *SafeBus* (`sbus`), *ServersAndClients* (`servers`), *SharedMemory* (`smemory`), *Sudoku* (`sudoku`), *CircularTrains* (`train`), and *TokenRing* (`tring`). The biggest model in terms of state space size is `philo20` (3.49E+09 reachable states). We were limited in our choice to Place-Transition 1-safe Petri nets because generalized Petri nets are not supported by `MISTA`. We ran all the experiments on a Dell Precision Tower 3430 with a processor Intel Xeon E-2136 6-cores @ 3.3GHz, 64 GiB memory, and Ubuntu 20.04. We used a timeout of 1 hour. The reader can find in the repository [24] all the files needed to reproduce the benchmarks and the figures.

Table 1 summarizes our results. For each model, we indicate the number of transitions (column 3), the number of observable transitions obtained by our structural analysis of the model (column 4), the number of obtained covering firing sequences (column 5) and their average size (column 6), and the execution time of the three tested tools (column 7) where T1, T2 and T3 refer to our approach (namely `sogMBT`), `MISTA` and `NModel`, respectively. It is clear that `NModel`

model	instance	# trans.	# obs. trans.	# paths			Average size of paths			Time (ms)		
				T1	T2	T3	T1	T2	T3	T1	T2	T3
example	example	11	4	2	3	4	8.00	7.00	9.00	**0.95**	1.0	100.0
philo	philo10	50	30	14	2	TO	39.07	73.00	TO	**1344.970**	1668.0	TO
	philo20	100	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
	philo5	25	15	9	2	150	12.44	33.50	8.77	7.57	**6.0**	337.0
referendum	referendum10	21	20	19	10	TO	10.53	11.00	TO	2622.98	**1785.0**	TO
	referendum15	31	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
sbus	sbus3	91	85	20	24	415	267.70	97.67	80.82	314.93	**179.0**	23522.0
	sbus6	451	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
servers	servers100-20	4200	2100	2000	2000	1	3.05	3.05	8000.00	335988.00	**1640.0**	34303.0
	servers100-40	8200	4100	4000	4000	TO	3.03	3.03	TO	1300630.00	**6054.0**	TO
	servers100-80	16200	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
smemory	smemory10	210	100	100	100	TO	8.75	45.20	TO	11475.00	**269.0**	TO
	smemory20	820	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
	smemory5	55	25	25	15	193	5.56	99.67	71.75	**38.49**	72.0	4271.0
sudoku	sudokuA-1	1	1	1	1	1	1.00	1.00	1.00	**0.68**	9.0	85.0
	sudokuA-2	8	8	4	3	32	3.00	3.00	3.00	0.919	**0.0**	113.000
	sudokuA-3	27	27	17	11	19494	7.26	5.73	6.06	**197.47**	416.0	52540.0
	sudokuA-4	64	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
train	train12	12	4	1	1	21	12.00	17.00	29.52	**2.40**	16.0	215.0
	train24	24	8	1	TO	TO	24.00	TO	TO	**330.64**	MO	TO
	train48	48	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
tring	tring10	1111	1111	106	46	TO	18.36	142.54	TO	68507.90	**9743.0**	TO
	tring15	3616	TO	TO	TO	TO	TO	TO	TO	TO	MO	TO
	tring5	156	156	11	11	100	10.18	59.27	6.87	9.89	**5.0**	261.0

T1: `sogMBT`; T2: `MISTA`; T3: `NModel`

Table 1. Experimental results (timeout 1 hr)

could not compete with the other two tools. Amongst the 24 situations, `MISTA`

[7] https://cosyverif.lipn.univ-paris13.fr/
[8] https://mcc.lip6.fr

outperformed `sogMBT` in 9 cases, while the reverse was true 7 times. It is worth noting that we do not expect our approach to be efficient when the number of observable transitions is high. This is the case for three out of 8 of the selected models: `referendum` (all the transitions, except one, are observable), and both `sudoku` and `tring` models where all the transitions are observable. If we focus on the testing effort that we can deduce from the two penultimate columns, which can be measured by the product of the number of paths and the average path size, we see that amongst 16 situations, the testing effort related to `sogMBT` is lower in 7 (resp. `MISTA` in 6) while three cases show perfect equality. `MISTA` uses a DFS strategy, leading to an average size of the generated paths longer than the computed by our approach.

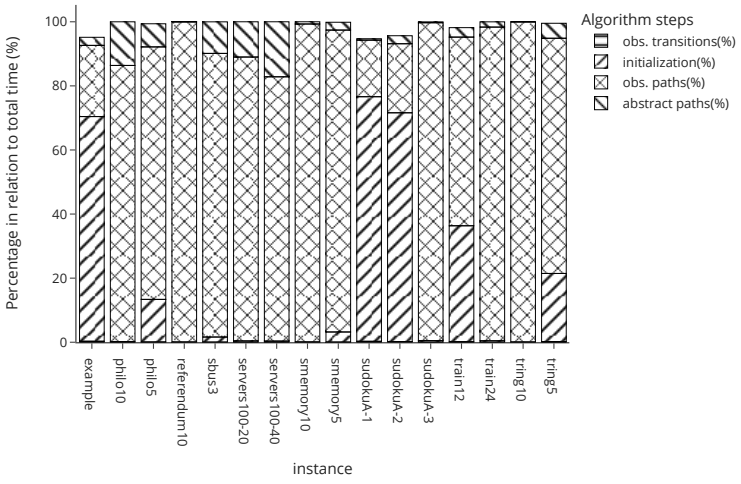

Fig. 5. Times for each step in the computation of observable traces. Some bars do not reach the 100% because some times are very small, thus accumulating precision errors.

Fig. 5 is given in order to have a detailed analysis of the distribution of the execution time of `sogMBT` among the different phases of our approach: (1) initialization of data structure (BDD staff); (2) computation of the observable transitions; (3) computation of observable paths; and (4) extraction of abstract paths from observable ones. It is clear that for small models, the initialization time is prevailing while for medium to large models, the generation time of the observed paths is more important since the construction of the aggregates is realized during the exploration. The last phase is not very time-consuming.

To summarize these preliminary results, the `sogMBT`, as a proof of concept prototype, is competitive w.r.t. existing tools. This is promising, and the approach can be improved from different perspectives. For instance, more sophisticated heuristics can be elaborated to decide which path to follow during the DFS-based construction/exploration of the SOG, in order to cover the observ-

able transitions with less effort. Another improvement would be to parallelize the initialization phase (e.g., following [23]), among others.

6 Related work

Given the widespread use of Petri nets in the specification of critical systems, several works have aimed at developing approaches for generating test inputs from models described with them. Many research projects have combined the reachability graph and standard search algorithms from graph theory to produce test inputs [29]. Other approaches have used the same structure to build a transition tree representing the test inputs. This transition tree is rooted in the initial state of the marking graph, and each path leading to a leaf is a sequence of firing transitions from the Petri net [47]. Dianxiang et al. [49] have shown that this data structure is helpful to have provided a strategy for robustness testing. Dianxiang [48] published an integrated development environment for automated test generation and execution two years earlier.

Identifying the change related to a specification modification has been the subject of two research projects on regression testing [1,16]. Their main goal was to identify test inputs which are no longer relevant due to the removal of some transitions. Besides, they propose new entries or modify old ones to cover the new transitions added to the model. The test quality measurement of the interactions between agents was introduced by Miller et al. [36]. Authors have modified an existing debugging tool to measure the coverage rate of various protocol-based criteria related expressed using Petri nets. Several coverage criteria have been formalized for high-level Petri nets [51,30]. The work of [14] uses the cause-effect network's concept for the generation of test inputs, while [15] uses the execution potential of the model by the simulation to measure the coverage rate of the generated test inputs. Most of these approaches face the problem of state space explosion when aiming to test systems with a marking graph containing thousands of states. From the work of Chusho [12], which introduces the notion of the essential branch, to that of Bardin et al. [6], which defines a unified framework to identify essential test objectives, several works and theories have emerged to address this problem. They rely on graph theory, dynamic symbolic execution, weakest calculus, model-checking, proof, constraint-based techniques, and value analysis. Based on the notion of the decision-to-decision path (*dd-path*) [21], Chusho [12] introduced the essential branch measure, which represents the cornerstone to transform a control flow graph (CFG), representing the target code, into a reduced graph called an inheritor-reduced graph. Following this reduction, covering essential branches implies the coverage of all branches. In the same vein, Bertolino and Marré [9] have used the relations of dominance and implication among the arcs of a *ddgraph*, which allowed the deduction of unconstrained arcs. The deduced set constitutes a minimal set of the ddgraph's arcs such that the paths which cover them are a path cover. These approaches cannot identify infeasible structural objectives without human assistance to tune the selection strategy. Also, these approaches are suitable to handle CFG but do not scale

up to handle large systems with thousands of nodes. A broader perspective of our work includes the generation of test paths such as those proposed by Li et al. [28]. They have used search-based approaches, whereas the one proposed by Hallé revolves around Cayley graph and triaging function. The latter defines a unified method to handle different coverage criteria, and the Cayley graph gathers the test paths into equivalence classes. As a result, this approach reduces the number of test paths. However, it does not offer a solution to identify the unfeasible objectives and the subsumed elementary components (transitions).

Offutt and Pan [39] encode the test objectives under constraints that are extracted from the program under test. Unresolved constraints represent infeasible objectives. By associating properties to the structural test objects, the use of model-checking [10] verifies whether the test object is verifiable or not. If the first approach shows limits when the constraints are non-linear [2] or when the program uses aliases [27], model-checking faces the problem of scaling.

Other techniques use the weakest precondition to remedy the scaling problem. This technique allows [7] to identify infeasible instructions while [4] determines infeasible branches. Since these first two criteria are the most basic, Bardin et al. [5] use the weakest precondition to address more advanced structural test objectives. Marcozzi et al. [31] unfold this technique to identify polluting test targets that do not stop at infeasible targets. Blending the identification of infeasible test objectives with symbolic or even concolic execution accelerates the generation of test inputs and refines the coverage measures.

7 Conclusion

Unlike verification, which can show that software is fault-free, testing discloses the presence of faults in the software only. Although researchers are not unanimous about the ability of test inputs that meet the coverage criteria to expose flaws, these criteria can help to evaluate the quality of the tests generated. We have formalized the separation of a Petri net's transitions into two disjoint sets: *Obs* and *UnObs*. We have shown that an *UnObs* transition is either unreachable or subsumed by another transition. Also, we have shown that the coverage of all the *Obs* transitions meets the transition coverage criterion of a given Petri net. Then, we have proposed a structural manner to extract the set of observable transitions from a Petri net. The construction of the *Obs* test paths was presented through an algorithm based on the partial building of a SOG. Yet, we have introduced the final algorithm, which builds abstract test paths from *Obs* tests. Since it is not mandatory to build the whole SOG's aggregates, our approach shows potential signs to improve the performance of test path generation. This has been confirmed by our preliminary experimental results. As future work, we plan to use linear programming and machine learning approaches in order to minimize the number and the length of the test sequences during the SOG construction. Also, we aim to extend our approach to other types of coverage criteria related to Petri nets and to evaluate the potential of fault identification using mutation testing.

References

1. Ahmad, F., Qaisar, Z.H.: Scenario based functional regression testing using petri net models. In: ICMLA (2). pp. 572–577. IEEE (2013)
2. Anand, S., Burke, E.K., Chen, T.Y., Clark, J.A., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.: An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. **86**(8), 1978–2001 (2013)
3. Arias, J., Bennani, M.T., Desel, J., Klai, K., Ochi, H.: sogMBT: Symbolic observation graph-based generator of test paths (2023), https://depot.lipn.univ-paris13.fr/PMC-SOG/sogMBT
4. Baluda, M., Denaro, G., Pezzè, M.: Bidirectional symbolic analysis for effective branch testing. IEEE Trans. Software Eng. **42**(5), 403–426 (2016)
5. Bardin, S., Delahaye, M., David, R., Kosmatov, N., Papadakis, M., Traon, Y.L., Marion, J.: Sound and quasi-complete detection of infeasible test requirements. In: ICST. pp. 1–10. IEEE Computer Society (2015)
6. Bardin, S., Kosmatov, N., Marcozzi, M., Delahaye, M.: Specify and measure, cover and reveal: A unified framework for automated test generation. Sci. Comput. Program. **207**, 102641 (2021)
7. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. IEEE Trans. Software Eng. **36**(4), 495–508 (2010)
8. Bernardino, M., Rodrigues, E.M., Zorzo, A.F., Marchezan, L.: Systematic mapping study on MBT: tools and models. IET Softw. **11**(4), 141–155 (2017)
9. Bertolino, A., Marré, M.: Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Trans. Software Eng. **20**(12), 885–899 (1994)
10. Beyer, D., Chlipala, A., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: ICSE. pp. 326–335. IEEE Computer Society (2004)
11. Chariyathitipong, P., Vatanawood, W.: Dynamic slicing of time petri net based on MTL property. IEEE Access **10**, 45207–45218 (2022)
12. Chusho, T.: Test data selection and quality estimation based on the concept of esssential branches for path testing. IEEE Trans. Software Eng. **13**(5), 509–517 (1987)
13. Darvas, D., Adiego, B.F., Vörös, A., Bartha, T., Viñuela, E.B., Suárez, V.M.G.: Formal verification of complex properties on PLC programs. In: FORTE. LNCS, vol. 8461, pp. 284–299. Springer (2014)
14. Desel, J., Oberweis, A., Zimmer, T., Zimmermann, G.: Validation of information system models: Petri nets and test case generation. In: ICSMC. vol. 4, pp. 3401–3406. IEEE (1997)
15. Ding, J., Argote-Garcia, G., Clarke, P.J., He, X.: Evaluating test adequacy coverage of high level petri nets using spin. In: AST. pp. 71–78. ACM (2008)
16. Ding, Z., Jiang, M., Chen, H., Jin, Z., Zhou, M.: Petri net based test case generation for evolved specification. Sci. China Inf. Sci. **59**(8), 1–25 (2016)
17. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: TACAS. LNCS, vol. 3440, pp. 382–396. Springer (2005)
18. Fernandez, J., Bozga, M., Ghirvu, L.: State space reduction based on live variables analysis. Sci. Comput. Program. **47**(2-3), 203–220 (2003)
19. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test. Verification Reliab. **19**(3), 215–261 (2009)

20. Haddad, S., Ilié, J., Klai, K.: Design and evaluation of a symbolic and abstraction-based model checker. In: ATVA. LNCS, vol. 3299, pp. 196–210. Springer (2004)
21. Huang, J.C.: Error detection through program testing. Current Trends in Programming Methodology **II**, 16–43 (1977)
22. Jr., S.B.A.: Binary decision diagrams. IEEE Trans. Computers **27**(6), 509–516 (1978)
23. Klai, K., Abid, C.A., Arias, J., Evangelista, S.: Hybrid parallel model checking of hybrid LTL on hybrid state space representation. In: VECoS. LNCS, vol. 13187, pp. 27–42. Springer (2021)
24. Klai, K., Bennani, M.T., Arias, J., Desel, J., Ochi, H.: sogMBT benchmarks (2023), https://depot.lipn.univ-paris13.fr/PMC-SOG/experiments/test-paths
25. Klai, K., Petrucci, L.: Modular construction of the symbolic observation graph. In: ACSD. pp. 88–97. IEEE (2008)
26. Klai, K., Poitrenaud, D.: MC-SOG: an LTL model checker based on symbolic observation graphs. In: Petri Nets. LNCS, vol. 5062, pp. 288–306. Springer (2008)
27. Kosmatov, N.: All-paths testgeneration for programs with internal aliases. In: ISSRE. pp. 147–156. IEEE Computer Society (2008)
28. Li, N., Li, F., Offutt, J.: Better algorithms to minimize the cost of test paths. In: ICST. pp. 280–289. IEEE Computer Society (2012)
29. Li, Y., Zhang, X., Zhang, Y., Guo, J., Rao, C.: A test cases generation method for atp. In: RSVT. pp. 21–26. ACM (2021)
30. Liu, Z., Liu, T., Cai, L., Yang, G.: Test coverage for collaborative workflow application based on petri net. In: CSCWD. pp. 213–218. IEEE (2010)
31. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: ICSE. pp. 456–467. ACM (2018)
32. Martin-Lopez, A., Arcuri, A., Segura, S., Ruiz-Cortés, A.: Black-box and white-box test case generation for restful apis: Enemies or allies? In: ISSRE. pp. 231–241. IEEE (2021)
33. Mateescu, R., Wijs, A.: Sequential and distributed on-the-fly computation of weak tau-confluence. Sci. Comput. Program. **77**(10-11), 1075–1094 (2012)
34. Meng, Y., Gay, G.: Understanding the impact of solver choice in model-based test generation. In: ESEM. pp. 22:1–22:11. ACM (2020)
35. Micskei, Z.: Model-based testing (MBT), http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html#tools
36. Miller, T., Padgham, L., Thangarajah, J.: Test coverage criteria for agent interaction testing. In: AOSE. LNCS, vol. 6788, pp. 91–105. Springer (2010)
37. Murata, T.: Petri nets: Properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
38. Nosrati, M., Haghighi, H., Vahidi-Asl, M.: Test data generation using genetic programming. Inf. Softw. Technol. **130**, 106446 (2021)
39. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verification Reliab. **7**(3), 165–192 (1997)
40. Offutt, J., Thummala, S.: Testing concurrent user behavior of synchronous web applications with petri nets. Softw. Syst. Model. **18**(2), 913–936 (2019)
41. Pelánek, R.: Fighting state space explosion: Review and evaluation. In: FMICS. LNCS, vol. 5596, pp. 37–52. Springer (2008)
42. van der Sanden, B., Geilen, M., Reniers, M.A., Basten, T.: Partial-order reduction for supervisory controller synthesis. IEEE Trans. Autom. Control. **67**(2), 870–885 (2022)
43. Shafique, M., Labiche, Y.: A systematic review of state-based test tools. Int. J. Softw. Tools Technol. Transf. **17**(1), 59–76 (2015)

44. Teodorov, C., Leroux, L., Drey, Z., Dhaussy, P.: Past-free[ze] reachability analysis: reaching further with dag-directed exhaustive state-space analysis. Softw. Test. Verification Reliab. **26**(7), 516–542 (2016)
45. Utting, M., Legeard, B., Bouquet, F., Fourneret, E., Peureux, F., Vernotte, A.: Recent Advances in Model-Based Testing, vol. 101 (2016)
46. Villalobos-Arias, L., Quesada-López, C., Martínez, A., Jenkins, M.: Model-based testing areas, tools and challenges: A tertiary study. CLEI Electron. J. **22**(1) (2019)
47. Wang, C.C., Pai, W., Chiang, D.J.: Using a petri net model approach to object-oriented class testing. In: ICSMC. vol. 1, pp. 824–828. IEEE (1999)
48. Xu, D.: A tool for automated test code generation from high-level petri nets. In: Petri Nets. LNCS, vol. 6709, pp. 308–317. Springer (2011)
49. Xu, D., Tu, M., Sanford, M., Thomas, L., Woodraska, D., Xu, W.: Automated security test generation with formal threat models. IEEE Trans. Dependable Secur. Comput. **9**(4), 526–540 (2012)
50. Zhang, J., Zhang, D., Huang, K.: A regular expression matching algorithm using transition merging. In: PRDC. pp. 242–246. IEEE Computer Society (2009)
51. Zhu, H., He, X.: A methodology of testing high-level petri nets. Inf. Softw. Technol. **44**(8), 473–489 (2002)

A Proof of Lemma 1

Proof. Let t be an unobservable transition, p be a place in $t^{\sim}$ and $t' \in p^{\bullet}$. Assume that there exists a firing sequence $\sigma.t'$. Given that place p is not marked initially and that no transition, except t, can produce a token in p, t must be fired before t' to produce the necessary token in p.

B Proof of Lemma 2

Proof. Let $(\mathcal{N}, m_0)$ be a marked Petri net, $t_0 \in \mathit{UnObs}$ be an unobservable transition, $p \in t_0^{\sim}$ be an output place of t_0 having t_0 as unique input transition, and $t_1 \in p^{\bullet}$ be an output transition of p. Then, by Lemma 1, $t_1 \triangleright t_0$. If t_1 is observable, then the lemma is proved. Else, by an iterative application of this reasoning, by the transitivity of the coverage relation $\triangleright$ and by the fact that the number of transitions in a Petri net is finite, we end on two possible cases: (1) there exists a transition $t_n \in \mathit{Obs}$ s.t. $t_n \triangleright t_{n-1} \triangleright \cdots \triangleright t_0$, or (2) t_0 belongs to an unobservable cycle, which proves the Lemma.

C Proof of Lemma 3

Proof. Let t be a transition of the cycle, and $p \in {}^{\bullet}t$ be the input place of t that belongs to the cycle. p is initially empty because the (unique) transition $t' \in {}^{\bullet}p$ belongs to the cycle and hence is unobservable. Thus, place p will never be marked because t' is dead by a successive application of Corollary 1 from t.

D Proof of Theorem 2

Proof. Assume that $\mathit{Obs}' \subset \mathit{Obs}$ is the smallest subset of transitions satisfying Theorem 1. Let $t \in \mathit{Obs} \setminus \mathit{Obs}'$, there exists then a transition $t' \in \mathit{Obs}'$ s.t. $t' \triangleright t$, i.e., $\forall$ firing sequence $t_1 \dots t_n.t'$ (for $n \geq 1$), $\exists i \in \{1, \dots, n\}$ s.t., $t = t_i$. Let $p \in t^{\bullet} \cap {}^{\bullet}t'$. Then, by Definition 7, p is necessarily not marked, otherwise the sequence $t_{i+1} \dots t_n.t'$ is a firing sequence that does not contain t. Also, there exists a transition $t'_i \neq t$ in ${}^{\bullet}p$, otherwise t is unobservable. If t'_i is observable, then there exists a firing sequence $\beta.t'_i$ which makes the sequence $\beta.t'_i.t_{i+1} \dots t_n.t'$ a firing sequence not involving t, and that is not possible. If t'_i is unobservable, then, by using Lemma 2, there exists a transition $t''_i \in \mathit{Obs}$ s.t. $t''_i \triangleright t'_i$. By hypothesis, there exists a sequence $\alpha.t'_i.\gamma.t''_i$. Thus, the sequence $\alpha.t'_i.t_{i+1} \dots t_n.t'$ covers t' but not t, which is impossible. To conclude, such a transition t could not exist and Obs' is necessarily equal to Obs.