SOG-Based Multi-Core LTL Model Checking

Chiheb Ameur Abid, Kais Klai, Jaime Arias, Hiba Ouni

SOG-Based Multi-Core LTL Model Checking

1st Chiheb Ameur Abid
Mediatron Lab
SupCom, University of Carthage Tunis, Tunisia
chiheb.abid@fst.utm.tn

2nd Kais Klai
LIPN, CNRS UMR 7030
University Sorbonne Paris North France
kais.klai@lipn.univ-paris13.fr

3rd Jaime Arias
LIPN, CNRS UMR 7030
University Sorbonne Paris North France
arias@lipn.univ-paris13.fr

4th Hiba Ouni
LIPN, CNRS UMR 7030
University Sorbonne Paris North France
hiba.ouni@lipn.univ-paris13.fr

***Abstract*—The model checking is one of the major techniques used in the formal verification. This technique builds on an automatic procedure that takes a model M of a system and a formula φ expressing a temporal property, and decides whether the system satisfies the property (denoted by $M \models \varphi$). The model checking technique is based on an exhaustive exploration of the state space of the system and, thus suffers from the state space explosion problem: it can happen that the verification process stops because of lack of time or space. Among the existing solutions to tackle this problem the Symbolic Observation Graph (SOG) has been proposed as a reduced representation of the reachability graph preserving linear temporal logic properties (LTL) i.e. checking an LTL property on the SOG is equivalent to check it on the original state space. The parallel construction of the SOG could increase the speedup and scalability of model checking. In this paper, we propose a new model checking algorithm built on a parallel construction of the SOG. The SOG is adapted to allow the preservation of both state and event-based LTL formulae i.e., the atomic propositions involved in the formula to be checked could be either state-based or event-based propositions. We implemented the proposed model checking algorithm within a C++ prototype and compared our preliminary results with the state of the art model checkers.**

***Index Terms*—Parallel model checking, Temporal Logic, Decision Diagrams.**

I. Introduction

Model checking [1] is a powerful formal verification method that can be used to improve the safety of concurrent systems. Given a formal specification, model checking algorithms analyze the system behavior by constructing/checking the whole reachable state space. The reachable state space is traversed to find error states that violate safety properties, or to find cyclic paths on which no progress is made as counterexamples for liveness properties. This technique builds on an automatic procedure that takes a model M of a system and a formula φ expressing a temporal property, and decides whether the system satisfies the property (denoted by $M \models \varphi$). The automata-based LTL verification decision procedure is reduced to the emptiness check of a synchronized product between two automata A_M and $A_{\neg\varphi}$, denoted by $A_M \times A_{\neg\varphi}$. A_M represents the state space of the system and $A_{\neg\varphi}$ represents the automaton of the negation of the formula φ to be verified (i.e. accepting all the words that do not satisfy φ). Since $A_M \times A_{\neg\varphi}$ accepts all system's executions that do not satisfy φ, its emptiness implies that all system's executions satisfy φ. Otherwise, any execution accepted by $A_M \times A_{\neg\varphi}$ represents a counter-example allowing the designer to refine his model and re-launch the model checking procedure.

The main limitation of model checking technique is the well-known problem of combinatorial state space explosion [2]. Indeed, the size of the state space of a given system grows exponentially with the number of its components. In practice, it can happen that the verification process stops because of lack of time or space. Several approaches have been proposed to cope with the state space explosion problem in order to get a manageable state space and to improve the scalability of the model checking. These approaches can roughly be classified in two large families, namely *explicit* and *symbolic* approaches.

Explicit model checking approaches explore an explicit representation of the synchronized product graph. A common optimisation builds the graph on-the-fly as required by the emptiness check algorithm: the construction stops as soon as a counterexample is found (e.g., [3]–[5]). *Partial order reduction* (e.g., [6]–[8]) is a reduction technique exploiting independence system's actions to discard unnecessary parts. Another source of optimisation is to take advantage of stuttering equivalence between paths in the state space graph when verifying a stuttering-invariant property [9]: this has been done either by ignoring some paths in the state space graph [10], or by representing the property using a *testing automaton* [11]. To our knowledge, all these solutions require dedicated algorithms to check the emptiness of the product graph.

Symbolic model checking tackles the state space explosion problem by representing the product automaton symbolically, usually by means of decision diagrams (a concise way to represent large sets or relations, e.g., BDDs [12], MDDs [13]). Various symbolic algorithms exist to verify LTL using fixpoint computations (see [14], [15] for comparisons and [16] for the clarity of the presentation). As-is, such approaches do not mix well with partial order, stuttering invariant reductions and on-the-fly emptiness check.

However, explicit and symbolic approaches are not exclusive, some combinations have already been studied [17]–[20]

to get the best of both worlds. They are referred to as **hybrid approaches**. Most of these approaches consist in replacing the state space of the system by an explicit graph where each node contains sets of states, that is an abstraction of the system graph preserving properties of the original system.

In addition to techniques for reduction and compression, parallel and distributed-memory processing can be used [21]. The use of distributed processing increases the speedup and scalability of model checking by exploiting the cumulative computational power and memory of a cluster of computers. Such approaches have been studied in various contexts leading to different proposed solutions for both symbolic and explicit model checking (e.g [21]–[25]).

In this paper, we propose a parallel model checking technique based on the Symbolic Observation Graph (SOG). The SOG is an abstraction of the reachability state graph of concurrent systems [18], [20]. A SOG is a graph whose construction is guided by a set of *observable* atomic propositions involved in a linear time temporal formula. These atomic propositions can represent events or actions (event-based SOG [18]) or state-based properties (state-based SOG [20]). The nodes of a SOG are aggregates hiding a set of local states which are equivalent with respect to the observable atomic propositions, and are compactly encoded using Binary Decision Diagram techniques (BDDs) [12]. The arcs of an event-based SOG are exclusively labeled with observable actions. It has been proven that both event and state-based SOGs preserve *stutter-invariant* LTL formulae ([18], [20]).

In previous works, we have investigated different approaches to parallelize the SOG construction using different algorithms to benefit from additional speedups and performance improvement in execution time and memory saving [26]–[29]. The key idea of our approaches is to build simultaneously several nodes (aggregates) of the symbolic graph.

In this paper, we exploit the strengths of the parallel exploration/construction of the SOG to design a parallel model checking, where both event- and state-based properties can be expressed, combined, and verified. Instead of composing the whole system with the Büchi automaton of the negation of the formula to be checked, we perform the synchronization of the automaton with an abstraction of the original reachability system's graph: an event/state-based SOG.

The event-based and state-based semantics are interchangeable: an event can be encoded as a change in state variables, and likewise one can equip a state with different events to reflect different values of its internal variables. However, converting from one representation to the other often leads to a significant enlargement of the state space. Typically, event-based semantics is adopted to compare systems according to some equivalence or pre-order relation (e.g. [30], [31]), while state-based semantics is more suitable to model-checking approaches [32]. Combining both semantics allows to express properties in a compact and intuitive manner.

In practice, due to the small number of atomic propositions in a typical LTL formula, the SOG has a very moderate size. Previous works have shown that the SOG-based approach is an interesting alternative for LTL model checking outperforming (in general) both explicit and purely symbolic verification approaches ([18], [20], [33], [34]).

In this paper, we propose model checking algorithms built on our parallel construction of the SOG. The main contributions of this paper are:

- The proposal of a parallel on-the-fly LTL model checker (PMC-SOG) based on parallel construction of the SOG.
- The extension of the SOG definition to handle both event- and state-based LTL properties.
- The design and evaluation of PMC-SOG.

The paper is structured as follows: First, we present the necessary formalisms, basic preliminaries and notations in Section II. Then, in Section III, we introduce the event and state-based SOG. Section IV describes the main contribution of the paper: a multi-core model checker based on the parallel construction of an event- and state-based SOG. The proposed approach is evaluated and compared to other related works in Section VI. Finally, Section VII is dedicated to conclusion and perspectives.

II. Preliminaries

This section is dedicated to the definition of some relevant concepts and to the presentation of useful notations.

A. Labeled Kripke Structures and Hybrid LTL

In this paper, we consider hybrid LTL formulae where both state- and event-based atomic propositions can occur. In consequence, we choose to represent the semantics (behavior) of a system by a *Labeled Kripke Structure* (LKS for short).

Definition 1 (Labeled Kripke structure): Let AP be a finite set of atomic propositions and let Act be a set of actions. An LKS over AP is a 5-tuple $\langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ where:

- Γ is a finite set of *states* ;
- $L : \Gamma \rightarrow 2^{AP}$ is a labeling (or interpretation) function;
- $\rightarrow \subseteq \Gamma \times Act \times \Gamma$ is a *transition relation* ;
- $s_0 \in \Gamma$ is the *initial state*.

Definition 2 (Hybrid LTL): Given a set of atomic propositions AP and a set of actions Act, an LTL formula is defined inductively as follows:

- each member of $AP \cup Act$ is a formula,
- if ϕ and ψ are LTL formulae, so are $\neg\phi$, $\phi \vee \psi$, $X\phi$ and $\phi U \psi$.

Other temporal operators e.g., F (eventually) and G (always) can be derived as follows: $F\phi = true \cup \phi$ and $G\phi = \neg F \neg \phi$.

An interpretation of an LTL formula is an infinite run $w = x_0 x_1 x_2 \ldots$ (of some LKS), assigning to each state a set of atomic propositions and a set of actions that are satisfied within that state. An atomic proposition $p \in AP$ is satisfied by a state s_i if it belongs to its label (i.e., $L(s_i)$) while an action $a \in Act$ is said to be satisfied within a state s_i if it occurs from this state in w (i.e., $(s_i, a, s_{i+1}) \in \rightarrow$). In our case (interleaving model of concurrency), where a single action can occur at a time, at most one action can be assigned to a state of a run. We write w^i for the suffix of w starting from x_i and $p \in x_i$,

for $p \in AP \cup Act$, when p is satisfied by x_i. The hybrid LTL semantics is then defined inductively as follows:

- $w \models p$ iff $p \in x_0$, for $p \in AP \cup Act$,
- $w \models \phi \vee \psi$ iff $w \models \phi$ or $w \models \psi$,
- $w \models \neg\phi$ iff not $w \models \phi$,
- $w \models X\phi$ iff $w^1 \models \phi$, and
- $w \models \phi U \psi$ iff $\exists i \geq 1$; $w^i \models \psi$ and $\forall 1 \leq j < i$, $w^j \models \phi$.

An LKS K satisfies an LTL formula φ, denoted by $K \models \varphi$ iff all its runs satisfy φ.

It is well known that LTL formulae without the *next operator* (X) are invariant under the so-called *stuttering equivalence* [35]. We use this equivalence relation to prove that event- and state-based SOGs preserves $LTL \setminus X$ properties. Stuttering occurs when the same atomic propositions hold on two or more consecutive states of a given path.

III. Event and State-Based SOG

We propose to adapt the symbolic observation graphs [18] in order to abstract systems' behavior while preserving hybrid LTL formulae. The Symbolic Observation Graph [18], [20], [36] is an abstraction of the reachability graph of concurrent systems. The construction of a SOG is guided by the set of atomic propositions occurring in the LTL formula to be checked. Such atomic propositions are called *observed* while the others are *unobserved*. Nodes of the SOG are called *aggregates*, each of them is a set of states encoded efficiently using decision diagram techniques. Despite the exponential theoretical complexity of the size of a SOG (a single state can belong to several aggregates), its size is much more reduced than the original reachability graph. The difference between the event- and the state-based versions of the SOG ([18] and [20], [36] respectively) is the aggregation criterion. In event-based version, observed atomic proposition correspond to some actions of the system and an aggregate contains states that are connected by unobserved actions. In state-based version, observed atomic propositions are Boolean state-based conditions and an aggregate regroups states with the same truth values of the observed atomic propositions.

In this section, we propose to define an event-state based SOG preserving hybrid LTL formulae (i.e., both state and action-based atomic propositions can be used within a same formula). The modeling framework consists of Labeled Kripke structures (LKS). The construction of the SOG depends on a set of actions Act and state variables appearing as atomic propositions AP involved by the formula to be checked.

A. Revisiting SOG for Hybrid LTL

The adaption of the SOG to hybrid LTL leads to a new aggregation criterium: (1) two states belonging to a same aggregate have necessarily the same truth values of the state-based atomic propositions of the formula, (2) For any state s in the aggregate, any state s', having the same truth values of the atomic propositions as s, and being reachable from s by the occurrence of an unobserved action, belongs necessarily to the same aggregate, and (3) for any state s in the aggregate, any state s' which is reachable from s by the occurrence of an observed action is necessarily not a member of the same aggregate (even if it has the same label as s), unless it is reachable from an other state s'' of the aggregate by unobserved action.

Definition 3 (Event-state based aggregate): Let $\mathcal{K} = \langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ be an LKS over a set of atomic propositions AP and let $Obs \subseteq Act$ be a set of observed actions of $\mathcal{K}$. An aggregate a of $\mathcal{K}$ w.r.t. Obs is a triplet $\langle S, d, l \rangle$ satisfying:

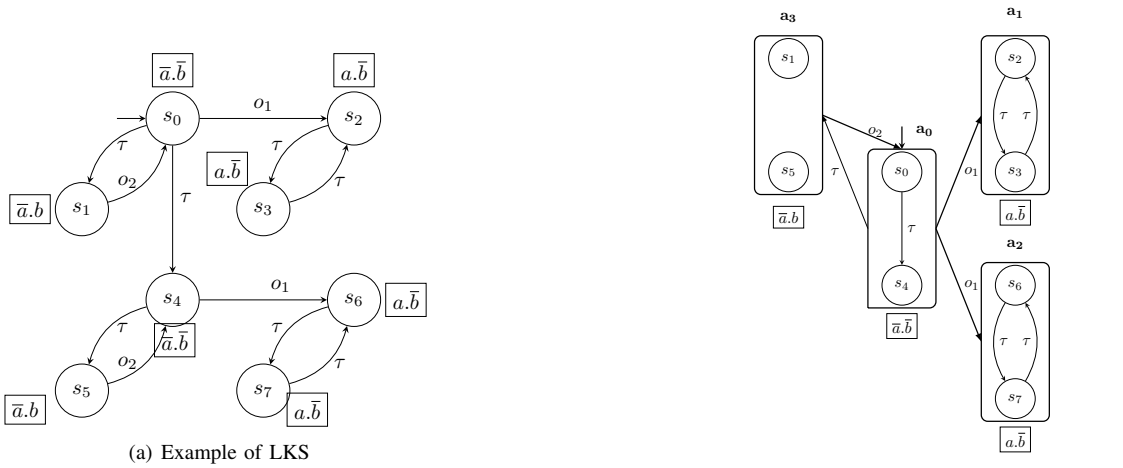
- $S \subseteq \Gamma$ where:
  - $\forall s, s' \in S$, $L(s) = L(s')$;
  - $\forall s \in S$, $(\exists (s', u) \in \Gamma \times (Act \setminus Obs) \mid L(s') = L(s) \wedge s \xrightarrow{u} s') \Rightarrow s' \in S$;
  - $\forall s \in S$, $(\exists (s', o) \in \Gamma \times Obs \mid s \xrightarrow{o} s') \wedge (\nexists (s'', u) \in S \times (Act \setminus Obs) \mid L(s'') = L(s') \wedge s'' \xrightarrow{u} s') \Rightarrow s' \notin S$.
- $d \in \{true, false\}$; $d = true$ iff S contains a dead state.
- $l \in \{true, false\}$; $l = true$ iff S contains an unobserved cycle (i.e., with unobserved actions).

Before defining the event- and state-based SOG, let us introduce the following operations:

- $SAT_{AP}(S)$: for a set of states $S \subseteq \Gamma$ with the same labels (i.e. such that $L(s) = L(s')$, for any $s, s' \in S$), returns the set of states that are reachable from any state in S, by a sequence of unobserved actions and which have the same value of the atomic propositions as S. It is defined as follows:
  $SAT_{AP}(S) = \{s'' \in \Gamma \mid \exists s \in S, \exists \sigma \in UnObs^*, s \xrightarrow{\sigma} s'' \wedge \forall s' \in \Gamma, \forall \beta \; prefix \; of \; \sigma, s \xrightarrow{\beta} s' \Rightarrow L(s) = L(s')\}$.
- $Out(a, t)$: returns, for an aggregate a and a action t, the set of states outside a that are reachable from some state in a by firing t. It is defined as follows:
  $$Out(a,t) \begin{cases} \{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{t} s'\} \; if \; t \in Obs \\ \{s' \in \Gamma \mid \exists s \in a.S, s \xrightarrow{t} s' \wedge L(s) \neq L(s')\} \\ if \; t \in UnObs \end{cases}$$
- $Out_\tau(a)$: returns, for an aggregate a, the set of states whose label is different from the label of any state of a, and which is reachable from some state in a by firing unobserved actions. It is defined as follows:
  $Out_\tau(a) = \bigcup_{t \in UnObs} Out(a, t)$.
- $Part_{AP}(S)$: returns, for a set of states $S \subseteq \Gamma$, the set of subsets of S that define the smallest partition of S according to the labeling function L. It is defined as follows:
  $Part_{AP} : 2^\Gamma \longrightarrow 2^{2^\Gamma}$
  $Part_{AP}(S) = \{S_1, S_2, \ldots, S_n\} \Leftrightarrow S = \bigcup_{i=1}^{n} S_i \wedge \forall i \in \{1..n\}, \forall s, s' \in S_i, L(s) = L(s') \wedge \forall s \in S_i, \forall s' \in S_j, j \neq i, L(s) \neq L(s')$.

Definition 4: Let $\mathcal{K} = \langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ be an LKS over a set of atomic propositions AP and let $Obs \subseteq Act$ be a set of observed actions of $\mathcal{K}$. **The SOG associated with $\mathcal{K}$, over AP and Obs**, is an LKS $\mathcal{G} = \langle A, Obs \cup \{\tau\}, L', \rightarrow', a_0 \rangle$ where:

1) A is a non empty finite set of aggregates satisfying :



(a) Example of LKS

(b) A corresponding SOG: $AP = \{a, b\}$ and $Obs = \{o_1, o_2\}$

Fig. 1. An LKS and its SOG

- $\forall a \in A,\ \forall t \in Obs, \forall o_i \in Part(Out(a,t)), \exists a' \in A\ s.t.\ a'.S = SAT_{AP}(o_i)$
- $\forall a \in A,\ \forall o_i \in Part(Out_\tau(a)), \exists a' \in A\ s.t.\ a'.S = SAT_{AP}(o_i)$

2) $L' : A \to 2^{AP}$ is a labeling (or interpretation) function s.t. $L'(a) = L(s)$ for $s \in a.S$;
3) $\to' \subseteq A \times Act \times A$ is the transition relation where:
   - $((a,t,a') \in \to') \Leftrightarrow ((t \in Obs) \wedge (\exists o_i \in Part(Out(a,t))$ s.t. $SAT_{AP}(o_i) = a')$
   - $((a,\tau,a') \in \to') \Leftrightarrow (\exists o_i \in Part(Out_\tau(a))$ s.t. $SAT_{AP}(o_i) = a')$
4) a_0 is the initial aggregate s.t. $s_0 \in a_0.S$.

The finite set of aggregates A of the SOG is defined in a complete manner such that the necessary aggregates are represented. The labeling function associated with a SOG gives to any aggregate the same label as its states. Point (3) defines the transition relation: (1) there exists an arc, labeled with an observed action t (resp. τ), from a to a' iff a' is obtained by saturation (using SAT_{AP}) on a set of equally labeled reached states $Out(a,t)$ (resp. $Out_\tau(a)$) by the firing of t (resp. any unobserved action) from $a.S$. The last point of Definition 4 characterizes the initial aggregate.

Figure 1(b) illustrates an event-state based SOG corresponding to the LKS of Figure 1(a). The presented SOG consists of 4 aggregates $\{a_0, a_1, a_2, a_3\}$ and 4 edges. The initial aggregate a_0 is obtained by adding any state reachable from the initial state s_0 of the LKS, by unobserved sequences of actions only, and labeled similarly to s_0. For this reason, the initial aggregate contains the state s_4. State s_2, which is reachable from s_0 by an observed action o_1, is excluded from a_0 and belongs to a_1. The same holds for s_6 which is reachable from s_4 by o_1 and belongs to the aggregate a_2. s_3 (resp. s_7) is added to a_1 (resp. a_2) since it is reachable from s_2 (resp. s_6) by an unobserved action and since it is labeled similarly. Note that one can merge a_1 and a_2 since they have the same label.

According to Definition 4, the SOG associated with an LKS is unique. It can also be non deterministic since, for instance, an aggregate can have several successors with τ (when the reached states, by τ, have different labels).

B. Checking stuttering invariant properties on SOGs

The equivalence between checking a given stuttering invariant formula (e.g., $LTL \setminus X$ formula) on the new adapted SOG and checking it on the original reachability graph is ensured by the preservation of maximal paths (finite paths leading to a dead state and infinite paths). First, the SOG preserves the observed traces of the corresponding model which allows to preserve infinite runs involving infinitely often observed transitions. Then, the truth value of the state-based atomic propositions occurring in the formulae are visible on the SOG by labeling each aggregate with the atomic propositions labeling (all) its states. Finally, the d and l attributes of each aggregate allow to detect deadlocks and livelocks (unobserved cycles) respectively. Note that the detection of the existence of dead states and cycles inside an aggregate is performed using symbolic operations (set operations) only.

In conclusion, the following result establishes that an LKS satisfies an $LTL \setminus X$ formula iff the corresponding SOG does.

Theorem 1: Let $\mathcal{K}$ be an LKS and let $\mathcal{G}$ be the corresponding SOG over Obs and AP. Let φ be an $LTL \setminus X$ formula on a subset of $Obs \cup AP$. Then $\mathcal{K} \models \varphi \Leftrightarrow \mathcal{G} \models \varphi$

The proof of this Theorem can be found at https://up13.fr/?azDbYg5n.

IV. Parallel LTL model checker based on the SOG

We propose an on-the-fly multi-core LTL model checking approach based on the SOG. This approach is intended for a configuration with shared memory architecture. We propose two versions for this approach by using two different techniques for the parallelization of the SOG building process. First used technique is based on the use of threads by following

the same approach presented in [26]. In this case, parallelism is performed at the level of construction of aggregates by creating a fixed number of threads such that every thread builds some aggregates. Second technique allows a finer parallelism granularity. Indeed, parallelization is performed at the level of the decision diagrams operations by using lock-less data structures and a work-stealing scheduling strategy.

Figure 2 illustrates different steps followed by the proposed model checker according to the two versions of parallelism. The proposed model checker considers an LKS and an LTL formula. After building the Büchi automaton of the formula negation, it initiates the parallel construction of the SOG simultaneously with the model checking process (computation of the synchronized product and the emptiness check). The model checker performs model checking in a sequential way while SOG construction is realized in parallel either by using a fixed number of threads to build several aggregates in parallel, or by parallelizing the construction of a single aggregate.

A. Parallelization at the level of aggregates

We recall that, in order to check a property that is expressed by an LTL formula on a given LKS, the parallel construction of the SOG is simultaneously performed with the model checking. We have two kinds of threads, namely *one model checker thread* and *a set of builder ones*. The model checker thread computes the synchronized product between the LKS representing the SOG and the Büchi automaton of the formula negation. It also performs the emptiness check of the product automaton on-the-fly. The builder threads cooperate simultaneously in order to build a SOG by adopting a dynamic load balancing scheme. For every thread a stack is associated in order to store the aggregates to be processed. A newly aggregate to be generated is pushed into the stack of the builder thread having the minimum load (i.e., the one having least elements in its stack).

Since the computation of the synchronized product and the building of the SOG are performed simultaneously, it is possible that the model checker thread tries to reach nodes of the SOG that are not yet built. For this reason, we add a Boolean attribute to every aggregate to indicate whether its successors are built, or not. By checking the value of this attribute of an aggregate, the model checker thread can check if it is possible to reach the successors of the aggregate, otherwise it has to wait until the attribute of the considered aggregate is updated.

The termination of the model checking algorithm is determined by the model checker thread. It is performed when the emptiness check process is finished, i.e. when the model checker thread terminates, builder threads are forcibly terminated by the model checker thread. The property is then proved to be unsatisfied by the system (an acceptance cycle has been found). If builder threads finish the construction of the SOG before the computation of the emptiness check is completed, only the builder threads terminate, while the model checker thread continues the exploration of the SOG until it determines the truth value of the property.

B. Parallelization at the level of decision diagrams operations

For this algorithm, a finer granularity level of parallelization is proposed for the construction of a SOG during model checking. We exploit parallel LDD operations already implemented in the Sylvan library [13], [37] which is used in the implementation of the SOG construction. The difference between the current and the previous algorithm is parallelism granularity. Indeed, in the previous algorithm, parallelization concerns aggregates construction, whereas the current algorithm performs parallelization at the level of LDD operations through the use of recursive functions. In the previous algorithm, only one thread is responsible of the construction of an aggregate that corresponds to one LDD tree. However, in the current approach, different threads cooperate in order to build one LDD tree that corresponds to an aggregate.

Sylvan is a multi-core decision diagrams library based on the work-stealing framework Lace [38]. Like the majority of work-stealing frameworks, Lace implements task-based parallelism by creating tasks (spawn) and waiting for their completion (sync) to use the results. Parallelization of LDD operations is performed by using lock-less data structures and work-stealing scheduling strategy. The basic idea behind work-stealing [39] is to break down a calculation into small tasks. Independent subtasks are stored in queues (work-pools) and idle processors steal tasks from the queues of busy processors. This will allow a processor to always have tasks to perform. The data structures used by Sylvan library are based on the lock-less paradigm, which ensures mutual exclusion and depends on atomic operations.

```
   Data: agg,dest : Aggregate;
   s: Stack;
   obs_tr : Set of actions;
1  obs_tr=getFirableObservablelactions(agg);
2  for every action t ∈ obs_tr do
3      s.push(t);
4      SPAWN(ComputeAggregate,get_successor(agg,t));
5  while s is not empty do
6      t=s.pop();
7      dest=SYNC(ComputeAggregate);
8      if dest does not exist in the SOG then
9          dest.div = isDiv(DEST);
10         dest.deadlock = isDeadlock(DEST);
11         Insert into the SOG the node dest;
12         Insert into the SOG the arc (agg,t,dest));
13     else
14     Insert into the SOG the arc (agg,t,dest)); ;
```

Algorithm 1: Aggregate successors using Lace

Consider an LTL formula and an LKS. The synchronized product, between the automaton modeling the negation of the formula and the SOG corresponding to the LKS is computed sequentially. However, building an aggregate is performed in parallel. The model checker starts by requesting the parallel

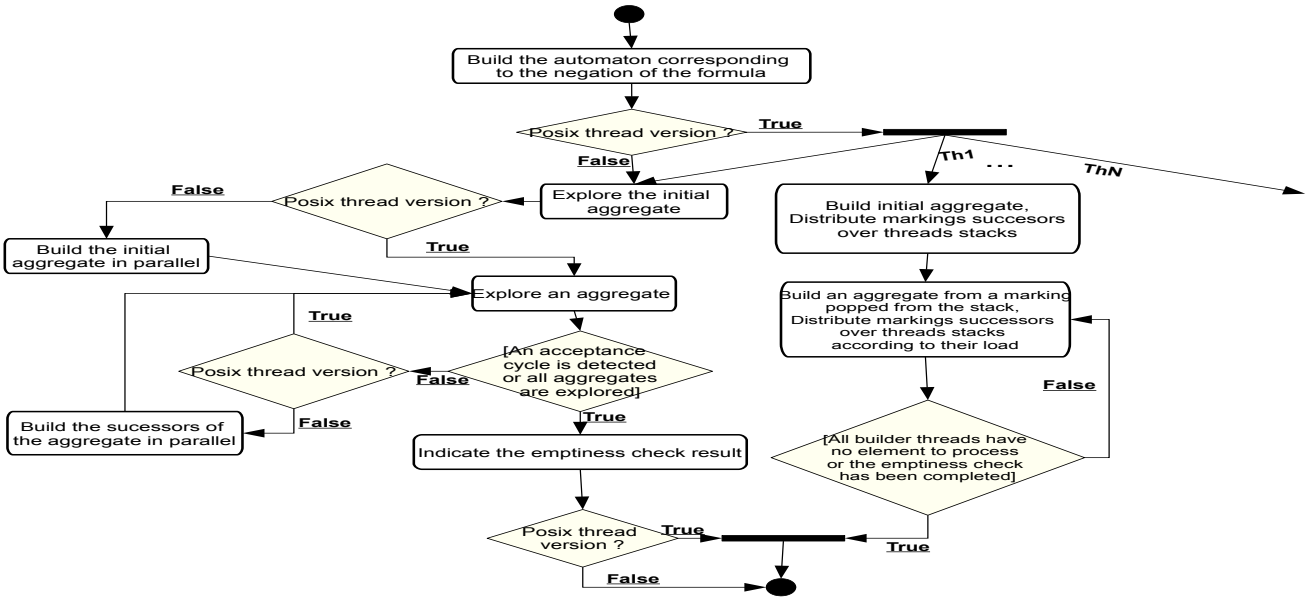


Fig. 2. Posix thread-based parallel LTL model checker using SOGs

construction of the initial aggregate. Then, it processes a loop in which it performs the computation of the synchronized product. In order to advance in the exploration of the SOG from an aggregate, the model checker triggers the construction of all successors of the current aggregate.

As illustrated by Algorithm 1, the construction of an aggregate successors is realized by using Sylvan library. It is initiated by computing enabled observable actions from the considered aggregate. Then, using $SPAWN$, for every enabled action t (t could be either an observed action, or an unobserved action changing the label of the current aggregate), we create a task for the recursive function $ComputeAggregate$ to compute the aggregate obtained by firing t. Results are retrieved by calling SPAWN. It is worth noting that $ComputeAggregate$ is in its turn implemented recursively by breaking it down into more small parallel tasks using $SPAWN$. Further, since tasks are stored in queues with a LIFO (Last In First Out) order, then last created task will be the first one to deliver its results. When a new aggregate is inserted in the SOG, we compute whether it contains a livelock (unobserved cycle) or a deadlock state. An aggregate that has a deadlock (resp.a lovelock) will correspond in the LKS built by the model checker to a node that has deadlock (resp. a livelock) successor that it is always reachable.

V. Implementation and Experimentation

A. Implementation

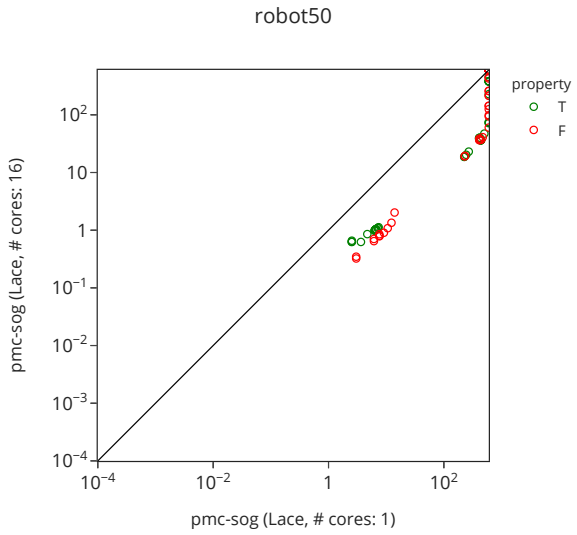
The implementation of the multi-core model checker is based on Spot library [40]. It is an object-oriented model checking library written in C++ that offers a set of building blocks that allow to develop LTL model checkers based on the automata theoretic approach. The chosen model of the system is a Petri net and the formula can involve places and/or transitions of the model.

The automaton class used by Spot to represent ω-automata is called action-based ω-automaton ($T\omega A$ for short). As its name implies, the $T\omega A$ class handles action-based acceptance, but it can emulate state-based acceptance using action-based acceptance by ensuring that all actions leaving an aggregate have the same acceptance set membership. In addition, there is a class, named $kripke$ that can be used to represent an LKS. During model checking, we translate built parts of the SOG to the LKS structure.

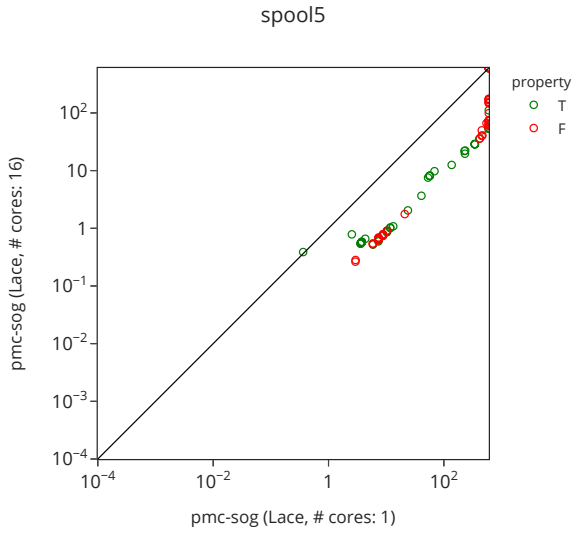
The checking algorithm visits the synchronized product of the ω-automaton corresponding to the negation of the formula and the LKS corresponding to the SOG. The translation of an LTL formula into an ω-automaton is proposed by Spot and it is dedicated to different formalism for the representation of the system to be checked.

Three abstract classes must be specialized to represent the ω-automata. The first abstract class defines a state, the second allows to iterate on the successors of a given state and the last one represents the whole ω-automaton. In our context, we have derived these classes for implementing a multi-core model checker based on the SOG. It is important to notice that the effective construction of the SOG is driven by the emptiness check algorithm of Spot and it can be managed on-the-fly. In our proposed approach, we have implemented the two aforementioned versions for the parallelization of the construction of the SOG.

In a given state, an atomic proposition associated with a place is satisfied if the place contains at least one token. In this case, the complete set of states corresponding to an aggregate is obtained by applying, until saturation, the transitions relation limited to the actions which do not modify the truth value



(a) Model `robot50`



(b) Model `spool5`

Fig. 3. Comparison of sequential and multi-core (16 cores) performance in `pmc-sog`

of atomic propositions. Instead of checking this constraint explicitly, we statically restrict the set of Petri net actions to be considered to the ones which do not modify the marking of the places involved in the formula to be checked.

VI. Experiments

The experimental results presented in this section were obtained on Magi cluster[1] of University Sorbonne Paris Nord. We used the partition `COMPUTE` which has 40 processors (two Intel Xeon E5-2650 v3 at 2.30GHz) connected by an InfiniBand network, and 64GB of RAM. A total of 5 models from the Model Checking Contest[2] were used in our experiments: *Philosophers* (`philo`), *RobotManipulation*(`robot`), *SwimmingPool* (`spool`), *CircularTrains* (`train`), and *TokenRing* (`ring`). The reader can find all the files needed to reproduce our experiments and figures at https://up13.fr/?azDbYg5n.

We exploit for these experiments a shared memory architecture with 24 cores. We measured the time (in seconds) consumed by the verification of 100 random formulas by progressively increasing the number of cores (1, 8, 16, 24). LTL\X formulas were generated by the tool `randltl`[3] and filtered into 50 satisfied and 50 violated properties. All the figures of this section are presented using a logarithmic scale. Each point represents a formula where satisfied properties are green and violated properties are red.

First, we compared the sequential and multi-core performance of our multi-core model checker (`pmc-sog`). Figure 3 shows the performance comparison when using 1 and 16 cores with models `robot50` and `spool5` as inputs. As we can observe, the multi-core execution outperforms the sequential one. In fact, the multi-core version manages to verify the formulas where the sequential version reaches a time limit. The same interpretation holds for all our experiments' results.

We then compared the two versions of `pmc-sog`: the one using POSIX threads (`pthreads` version) and the one using the work-stealing framework LACE (`lace` version). Figure 4 shows the comparison of both versions using 16 cores with models `robot50` and `spool5` as inputs. As we can observe, the version with POSIX threads outperforms the version using parallel LDD operations.

Finally, we performed a comparison between our tool (`pthreads` version) and the LTSmin model checker [41]. LTSmin[4] is an $LTL/CTL/\mu-calculus$ model checker that started out as a generic tool set for manipulating labelled transition systems. It accepts inputs in different modeling formalisms, *e.g.* PNML, UPPAAL, DiVinE. For sake of simplicity, we choose the traditional place/transition Petri nets in PNML format, thus we run `pnml2lts-mc` in our experiments since it is the LTSmin frontend that performs LTL model checking for PNML models. We also adopted the DFS (Depth First Search) algorithm for all approaches. Moreover, we considered only the atomic propositions based on the states because LTSmin is a state based model checker.

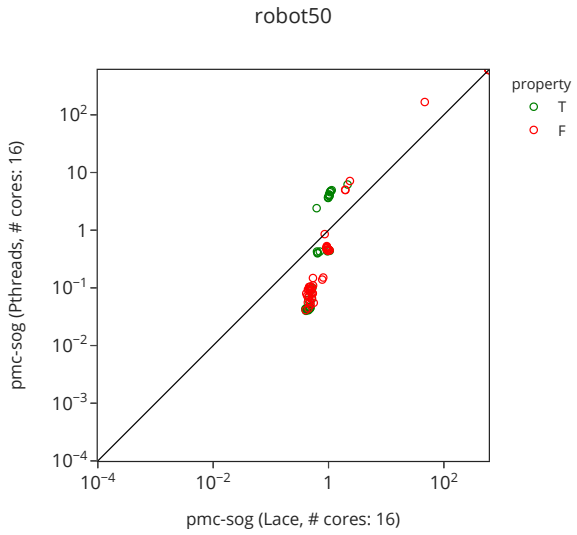
We keep all the parameters across the different model checkers the same. Tuning these parameters on a per-model basis could give faster results than presented here. It would, however, say little about the scalability of the core algorithms. Therefore, we decided to leave all parameters the same for all the models. We avoid resizing of the state storage in all cases by increasing the initial hash table size enough for all benchmarked input models.

[1]http://magi.univ-paris13.fr/wiki/

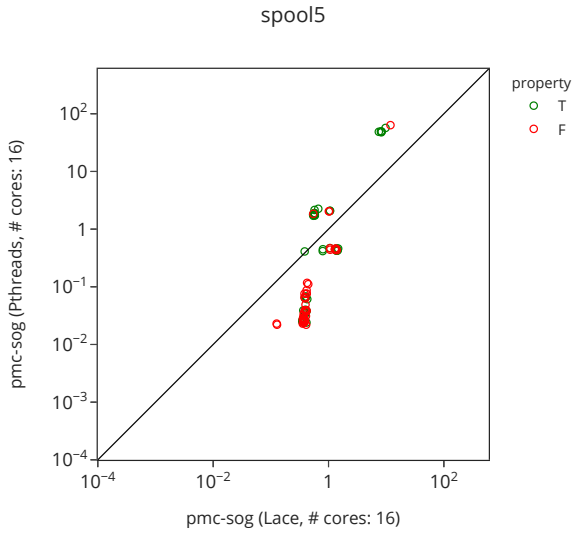
[2]https://mcc.lip6.fr/models.php

[3]https://spot.lrde.epita.fr/randltl.html

[4]https://ltsmin.utwente.nl/

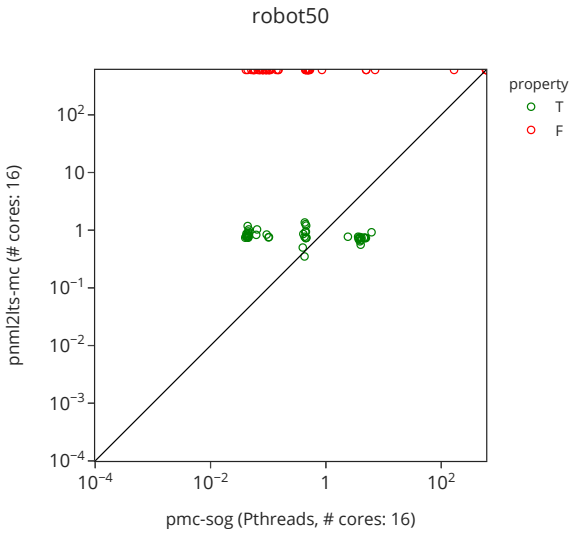


(a) Model `robot50` using 16 cores

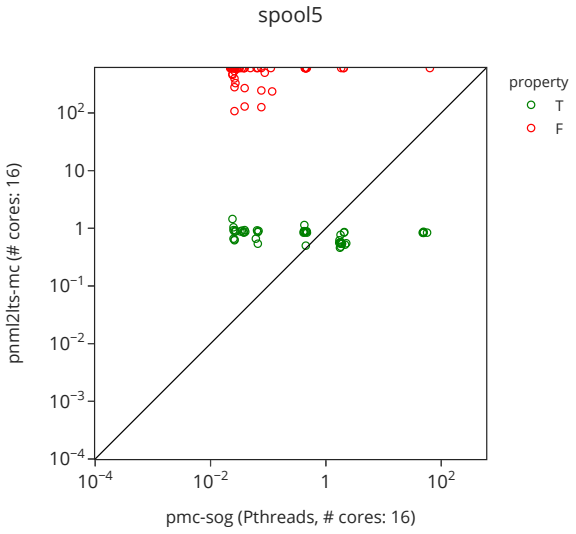


(b) Model `spool5` using 16 cores

Fig. 4. Comparison of `lace` and `pthreads` versions of `pmc-sog`



(a) Model `robot50`



(b) Model `spool5`

Fig. 5. Comparison of `pmc-sog` and `pnml2lts-mc` using 16 cores

Figure 5 shows a selection of our experimental results. As we can observe, our approach performs better than `pnml2lts-mc` for several experiments, especially unsatisfied properties. However, no model checker has an absolute advantage over the other for all the experiments: our model checker is the fastest for some models while LTSmin performs better for other cases. It is important to emphasize that our tool is still a prototype and better results could be found in a more mature version.

VII. Conclusion

In this paper, we proposed an on-the-fly multi-core model-checker approach for $LTL\setminus X$ logic based on event-based and state-based symbolic observation graphs. We have proposed two versions using different techniques of parallelization. One is based on the POSIX threads, and the other is based on the work-stealing framework LACE. The emptiness check is performed on-the-fly during the construction of the SOG allowing to reduce the runtime of the model checking process. Experiments show that our approach is competitive in com-

parison with the LTSmin although more experiments have to be achieved in order to compare the efficiency of our approach on other significant models and against other model checker tools. Also, we plan to check the behavior of our approach with a BFS-based emptiness check algorithm. Finally, it would be interesting to have a fully parallel approach where the emptiness check process is also parallel (e.g., [42]).

REFERENCES

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[2] A. Valmari, "The state explosion problem," in *Advanced Course on Petri Nets*. Springer, 1996, pp. 429–528.

[3] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis, "Memory-efficient algorithm for the verification of temporal properties," in *Proc. of CAV'90*, ser. LNCS, vol. 531. Springer, 1990, pp. 233–242.

[4] T. A. Henzinger, O. Kupferman, and M. Y. Vardi, "A space-efficient on-the-fly algorithm for real-time model checking," in *CONCUR*, ser. LNCS, vol. 1119. SV, 1996, pp. 514–529.

[5] J.-M. Couvreur, "On-the-fly verification of linear temporal logic," in *FM*, ser. LNCS, vol. 1709. SV, 1999, pp. 253–271.

[6] P. Godefroid and P. Wolper, "A partial approach to model checking," in *Proceedings of Annual IEEE Symposium on Logic in Computer Science, LICS'91*. IEEE, 1991, pp. 406–415.

[7] G. Bhat and D. Peled, "Adding partial orders to linear temporal logic," in *CONCUR*, ser. LNCS, vol. 1243. SV, 1997, pp. 119–134.

[8] A. Valmari, "A stubborn attack on state explosion," *Formal Methods in System Design*, vol. 1, no. 4, pp. 297–322, 1992.

[9] K. Etessami, "Stutter-invariant languages, ω-automata, and temporal logic," in *Proc. of CAV'99*, ser. LNCS, vol. 1633. Springer, 1999, pp. 236–248.

[10] R. Kaivola and A. Valmari, "The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic," in *Proc. of CONCUR'92*, ser. LNCS, vol. 630. Springer, 1992, pp. 207–221.

[11] H. Hansen, W. Penczek, and A. Valmari, "Stuttering-insensitive automata for on-the-fly detection of livelock properties," in *Proc. of FMICS'02*, ser. Electronic Notes in Theoretical Computer Science, vol. 66(2). Elsevier, 2002.

[12] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 293–318, 1992.

[13] T. van Dijk and J. van de Pol, "Sylvan: Multi-core decision diagrams," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 677–691.

[14] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang, "Is there a best symbolic cycle-detection algorithm?" in *Proc. of TACAS'01*, ser. LNCS, vol. 2031. Springer, 2001, pp. 420–434.

[15] F. Somenzi, K. Ravi, and R. Bloem, "Analysis of symbolic SCC hull algorithms," in *Proc. of FMCAD'02*, ser. LNCS, vol. 2517. Springer, 2002, pp. 88–105.

[16] Y. Kesten, A. Pnueli, and L. on Raviv, "Algorithmic verification of linear temporal logic specifications," in *Proc. of ICALP'98*, ser. LNCS, vol. 1443. Springer, 1998, pp. 1–16.

[17] A. Biere, E. M. Clarke, and Y. Zhu, "Multiple state and single state tableaux for combining local and global model checking," in *Proc. of CSD'99*, ser. LNCS, vol. 1710. Springer, 1999, pp. 163–179.

[18] S. Haddad, J.-M. Ilié, and K. Klai, "Design and evaluation of a symbolic and abstraction-based model checker," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2004, pp. 196–210.

[19] R. Sebastiani, S. Tonetta, and M. Y. Vardi, "Symbolic systems, explicit properties: on hybrid approches for LTL symbolic model checking," in *Proc. of CAV'05*, ser. LNCS, vol. 3576. Springer, 2005, pp. 350–363.

[20] K. Klai and D. Poitrenaud, "Mc-sog: An ltl model checker based on symbolic observation graphs," in *International Conference on Applications and Theory of Petri Nets*. Springer, 2008, pp. 288–306.

[21] J. Barnat, V. Bloemen, A. Duret-Lutz, A. Laarman, L. Petrucci, J. van de Pol, and E. Renault, "Parallel model checking algorithms for linear-time temporal logic," in *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pp. 457–507.

[22] G. J. Holzmann, "Parallelizing the spin model checker," in *International SPIN Workshop on Model Checking of Software*. Springer, 2012, pp. 155–171.

[23] I. Filippidis and G. J. Holzmann, "An improvement of the piggyback algorithm for parallel model checking," in *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. ACM, 2014, pp. 48–57.

[24] J. Barnat, L. Brim, and P. Rockai, "Divine 2.0: High-performance model checking," in *2009 International Workshop on High Performance Computational Systems Biology (HiBi 2009)*, 2009, pp. 31–32.

[25] J. Barnat, L. Brim, V. Havel, J. Havlíček, J. Kriho, M. Lenčo, P. Ročkai, V. Štill, and J. Weiser, "Divine 3.0–an explicit-state model checker for multithreaded c & c++ programs," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 863–868.

[26] H. Ouni, K. Klai, C. A. Abid, and B. Zouari, "A parallel construction of the symbolic observation graph: the basis for efficient model checking of concurrent systems," in *SCSS 2017. The 8th International Symposium on Symbolic Computation in Software Science 2017*, ser. EPiC Series in Computing, vol. 45, 2017, pp. 107–119.

[27] ——, "Parallel symbolic observation graph," in *Ubiquitous Computing and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on*. IEEE, 2017, pp. 770–777.

[28] ——, "Reducing time and/or memory consumption of the sog construction in a parallel context," in *Ubiquitous Computing and Communications (ISPA/IUCC), 2018 IEEE International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2018.

[29] ——, "Towards parallel verification of concurrent systems using the symbolic observation graph," in *2019 19th International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2019, pp. 23–32.

[30] Z. Tao, G. von Bochmann, and R. Dssouli, "Verification and diagnosis of testing equivalence and reduction relation," in *Proceedings of International Conference on Network Protocols*. IEEE, 1995, pp. 14–21.

[31] R. Kaivola and A. Valmari, "The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic," in *International Conference on Concurrency Theory*. Springer, 1992, pp. 207–221.

[32] J. Geldenhuys and A. Valmari, "Techniques for smaller intermediary bdds," in *International Conference on Concurrency Theory*. Springer, 2001, pp. 233–247.

[33] K. Klai and L. Petrucci, "Modular construction of the symbolic observation graph," in *ACSD*. IEEE, 2008, pp. 88–97.

[34] A. Duret-Lutz, K. Klai, D. Poitrenaud, and Y. Thierry-Mieg, "Self-loop aggregation product - a new hybrid approach to on-the-fly ltl model checking," in *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011.*, ser. Lecture Notes in Computer Science, vol. 6996, 2011, pp. 336–350.

[35] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000.

[36] K. Klai, S. Tata, and J. Desel, "Symbolic abstraction and deadlock-freeness verification of inter-enterprise processes," *Data & Knowledge Engineering*, vol. 70, no. 5, pp. 467–482, 2011.

[37] T. van Dijk and J. van de Pol, "Sylvan: multi-core framework for decision diagrams," *International Journal on Software Tools for Technology Transfer*, pp. 1–22, 2016.

[38] T. van Dijk and J. C. van de Pol, "Lace: non-blocking split deque for work-stealing," in *European Conference on Parallel Processing*. Springer, 2014, pp. 206–217.

[39] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[40] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu, "Spot 2.0—a framework for ltl and $\backslash\omega$-automata manipulation," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2016, pp. 122–129.

[41] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk, "Ltsmin: high-performance language-independent model checking," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 692–707.

[42] S. Evangelista, L. M. Kristensen, and L. Petrucci, "Multi-threaded explicit state space exploration with state reconstruction," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2013, pp. 208–223.

APPENDIX

A. Proof of Theorem 1

To prove Theorem 1, we will prove that the SOG preserves the maximal paths of the corresponding LKS. We recall that maximal paths (of finite systems) are any path π satisfying one of the following requirements:

1) $\pi = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} s_n$ such that s_n is a dead marking
2) $\pi = s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_l} s_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} s_n$ such that $s_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} s_n$ is a circuit.

Before giving the proof of the preservation of maximal paths, let us present two lemmas about the correspondence between paths of $\mathcal{K}$ and those of $\mathcal{G}$.

Lemma 1: Let $\pi = s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n$ be a path of $\mathcal{N}$ and a_1 be an aggregate of $\mathcal{G}$ such that $s_1 \in a_1$. Then, there exists a path $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t'_l} a_l$ of $\mathcal{G}$ and a strictly increasing sequence of integers $i_1 = 1 < i_2 < \cdots < i_{l+1} = n+1$ satisfying $\{s_{i_k}, s_{i_k+1}, \cdots, s_{i_{k+1}-1}\} \subseteq a_k$ for all $1 \leq k \leq l$.

Proof 1: We proceed by induction on the length of π. If $n = 1$, knowing that $s_1 \in a_1$ concludes the proof. Let $n > 1$ and assume that $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t'_{l-1}} a_{l-1}$ and $i_1, \cdots, i_l$ correspond to the terms of the lemma for the path $s_1 \xrightarrow{t_2} s_2 \xrightarrow{t_3} \cdots \xrightarrow{t_{n-1}} s_{n-1}$. Then, $s_{n-1} \in a_{l-1}$. Let us distinguish two cases.

(i) If $t_n \in UnObs \wedge L(s_{n-1}) = L(s_n)$ then, by definition of aggregates, $s_n \in a_{l-1}$. Thus both the path $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t'_{l-1}} a_{l-1}$ and the sequence $i_1, \cdots, i_l$ for the path of length $n-1$ stand for the path of length n as well.

(ii) If $t_n \in Obs \vee L(s_{n-1}) \neq L(s_n)$ then, since $s_{n-1} \xrightarrow{t_n} s_n$, there exists (by definition of the SOG) an aggregate a_l such that $a_{l-1} \xrightarrow{t_n} a_l$ and $s_n \in a_l$. As a consequence, the path $a_1 \xrightarrow{t'_2} a_2 \xrightarrow{t'_3} \cdots \xrightarrow{t_{l-1}} a_{l-1} \xrightarrow{t'_l = t_n} a_l$ and the sequence $i_1, \cdots, i_l, i_l + 1$ satisfy the proposition.

The next lemma shows that the inverse also holds.

Lemma 2: Let $\pi = a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} a_n$ be a path of $\mathcal{G}$. Then, there exists a path $s_1 \xrightarrow{\sigma_1} s'_1 \xrightarrow{t_2} s_2 \xrightarrow{\sigma_2} s'_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n \xrightarrow{\sigma_n} s'_n$ of $\mathcal{N}$ s.t., $\forall i = 1 \ldots n$, $\sigma_i \in UnObs^*$, s_i, s'_i belong to $a_i.S$ and all the traversed states from s_i and s'_i by firing σ_i have the same label (the same truth value of state-based atomic propositions).

Proof 2:

Let $\pi = a_1 \xrightarrow{t_2} a_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} a_n$ be a path of $\mathcal{G}$. First, let us define the four following functions related to π and any aggregate a_i of π, for $i = 1 \ldots n$.

- $In_\pi(a_i) = \begin{cases} \{s' \in a_i.S \mid \exists s \in a_{i-1} : s \xrightarrow{t_i} s'\} & \text{if } i \neq 1; \\ Out_\pi(a_i) & \text{otherwise.} \end{cases}$
- $Out_\pi(a_i) = \begin{cases} \{s \in a_i.S \mid s \xrightarrow{t_i}\} & \text{if } i \neq n; \\ In_\pi(a_i) & \text{otherwise.} \end{cases}$
- $Candidate_\pi(a_i) = \begin{cases} \{s \in In_\pi(a_i) \mid \exists s' \in Candidate'(a_i), \\ \exists \sigma \in UnObs^* : s \xrightarrow{\sigma} s'\} \text{ if } i \neq 1; \\ Candidate'_\pi(a_i) & \text{otherwise.} \end{cases}$
- $Candidate'_\pi(a_i) = \begin{cases} \{s \in Out_\pi(a_i) \mid \\ \exists s' \in Candidate(a_{i+1}),\ s \xrightarrow{t_{i+1}} s'\} \\ \text{if } i \neq n; \\ Candidate_\pi(a_i) \text{ otherwise.} \end{cases}$

Informally, $In_\pi(a_i)$ (for $i = 2 \ldots n$) represents the set of *input* states of aggregate a_i that are immediately reached by firing t_i from states in a_{i-1}. All the states of a_i are then obtained by adding the successors of this set of states by unobservable sequences, while having the same truth values of the state-based atomic propositions. For the first aggregate, $In_\pi(a_1)$ is the set of *output* states $Out_\pi(a_1)$ i.e., the states in $a_1.S$ enabling t_2. The same holds for $Out_\pi(a_i)$, for any $i = 1 \ldots n-1$ i.e., it contains states enabling action t_{i+1}. For a_n, $Out_\pi(a_n) = In_\pi(a_n)$.

Sets $Candidate_\pi(a_i)$ and $Candidate'_\pi(a_i)$, for $i = 1 \ldots n$ represent sets of states from which the states s_i and s'_i could be chosen, respectively, in order to build the path $s_1 \xrightarrow{\sigma_1} s'_1 \xrightarrow{t_2} s_2 \xrightarrow{\sigma_2} s'_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n \xrightarrow{\sigma_n} s'_n$ in a reverse order (starting from the end). In fact, $s'_n = s_n$ can be first chosen from $Candidate_\pi(a_n) = In_\pi(a_n)$. Then s'_{n-1} is obtained from $Candidate'_\pi(a_{n-1})$ i.e. states in $a_{n-1}.S$ enabling t_n and thus leading to $Candidate_\pi(a_n)$. s_{n-1} can be chosen from $Candidate_\pi(a_{n-1})$ i.e. to reach $Candidate'_\pi(a_{n-1})$ by unobservable actions only (and without traversing a differently labeled state), ... and so on.

We are now in position to study the correspondence between maximal paths to prove Theorem 1 through two new lemmas.

Lemma 3: Let $\pi = s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_n$ be a maximal path of $\mathcal{K}$. Then, there exists a maximal path $\pi' = a_0 \xrightarrow{t'_1} \cdots \xrightarrow{t'_l} a_l$ of $\mathcal{G}$ such that there exists a sequence of integers $i_0 = 0 < i_1 < \cdots < i_{l+1} = n+1$ satisfying $\{s_{i_k}, s_{i_k+1}, \cdots, s_{i_{k+1}-1}\} \subseteq a_k$ for all $0 \leq k \leq l$.

Proof 3: If s_n is a dead marking then knowing that $s_0 \in a_0$ and using Lemma 1, we can construct a path $\pi' = a_0 \xrightarrow{t'_1} a_1 \cdots \xrightarrow{t'_l} a_l$ and the associated integer sequence corresponding to π. Because the last visited state of π belongs to a_l, the dead attribute of a_l is necessarily equal to true and π' is then a maximal path of the SOG.

Now, if s_n is not a dead marking then, one can decompose π as follows: $\pi = \pi_1 \pi_2$ s.t. $\pi_1 = s_0 \xrightarrow{t_1} s_1 \rightarrow \cdots \xrightarrow{t_{k-1}} s_k$ and $\pi_2 = s_k \xrightarrow{t_{k+1}} \cdots \xrightarrow{t_n} s_k$ s.t., π_2 is a circuit. Once again, applying Lemma 1 from s_0, one can construct a path $\pi'_1 = a_0 \xrightarrow{t'_1} a_1 \xrightarrow{t'_2} \cdots a_m$ corresponding to π_1. The path in $\mathcal{G}$ associated with π'_2 can be also constructed applying the same lemma. However, this path must be constructed from a_m to which belongs s_k. π_1 and π_2 can be chosen as follows:

1) If π_2 involves unobserved actions and all traversed states (by π_2) have the same truth value of state-based atomic propositions, then π_2 is a cycle inside aggregate a_m and the *live* attribute (cycle) of a_m is necessarily set to true.
2) otherwise (i.e. either π_2 involves observed actions, or unobserved actions that change the truth value of state-based atomic propositions), then we chose the subpaths such that t_{k+1} as an observed action, or an unobserved one s.t. $(L)(s_k) \neq (L)(s_{k+1})$. By definition of the SOG,

since $s_k \in a_m$, there exists an aggregate a_{o_1} successor of a_m by action t_{k+1}. By using Lemma1, and the definition of the SOG, let $\pi'_1 = a_0 \xrightarrow{t'_1} a_1 \xrightarrow{t'_2} \cdots a_m$ and $\pi'_2 = a_{o_1} \xrightarrow{t_{p_1}} \ldots \xrightarrow{t_l} a_{q_1}$ with $s_k \in a_{q_1}.S$. If $a_{q_1} \xrightarrow{t_{k+1}} a_{o_1}$, then π'_2 is a circuit of $\mathcal{G}$ and $\pi'_1\pi'_2$ is a maximal path of $\mathcal{G}$ satisfying the proposition. Otherwise, by construction of the SOG, there exists an other successor of a_{q_1} containing s_k. Applying again Lemma 1 from this aggregate, we can construct a new path in $\mathcal{G}$ corresponding to π_2. Let $a_{o_2} \xrightarrow{t'_{p_1}} \ldots a_{q_2}$ be this path. If we can deduce a circuit of $\mathcal{G}$ from this path (if $a_{q_2} \xrightarrow{t_{k+1}} a_{o_2}$), this concludes the proof. Otherwise, we can construct a new path corresponding to π_2 starting from a successor of a_{q_2}. Because the number of aggregates in $\mathcal{G}$ is finite, in particular the number of aggregates to which belongs s_k is bounded by 2^N (where N is the number of state in the original LKS), a circuit will be necessarily obtained.

Notice that for all the previous cases above, a sequence of integers can be easily constructed from the ones produced by Lemma 1.

Lemma 4: Let $\pi' = a_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} a_n$ be a maximal path of $\mathcal{G}$. Then, there exists a maximal path $s_0 \xrightarrow{\sigma_1} s'_1 \xrightarrow{t_2} s_2 \xrightarrow{\sigma_2} s'_2 \xrightarrow{t_3} \cdots \xrightarrow{t_n} s_n \xrightarrow{\sigma_n} s'_n$ of $\mathcal{K}$ s.t., $\forall i = 1 \ldots n$, $\sigma_i \in UnObs^*$ and s_i, s'_i belong to $a_i.S$.

Proof 4: Let π' be a maximal path reaching an aggregate a_n such that $a_n.d = true \vee a_n.l$ (either the dead or the livelock attribute (cycle) is true). First, let us notice that the proof is trivial if the path π' is reduced to a single aggregate because dead state (resp. a state containing a circuit of a_0) is necessarily reachable from s_0.

Otherwise, using the same principle of Lemma 2 proof, one can demonstrate the existence of the maximal path in $\mathcal{K}$. We have just to define $In_\pi(a_0)$ as the singleton $\{s_0\}$ and $Out_\pi(a_n)$ as the dead state (if $a_n.d = true$) or the set of states forming a cycle in a_n (if $a_n.l = true$).

Now, if neither $a_n.d$ nor $a_n.l$ is true, then by construction of the SOG, $\pi' = a_0 \xrightarrow{t_1} \cdots \xrightarrow{t_l} a_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} a_n$ with $a_l \xrightarrow{t_{l+1}} \cdots \xrightarrow{t_n} a_n$ a circuit of $\mathcal{G}$ i.e., $a_n = a_l$. Here also, we can use the same scheme as for the proof of Lemma 2 by defining $In_\pi(a_0)$ as the singleton $\{s_0\}$ and $Out_\pi(a_n)$ as the set of states in a_n enabling t_{l+1}. Thus, starting from these states, and using the functions $Candidate_\pi()$ and $Candidate'_\pi()$ as defined in Lemma 2, one can build by backtracking the maximal path in $\mathcal{K}$ satisfying the terms of Lemma 4.