Hybrid Parallel Model Checking of Hybrid LTL on Hybrid State Space Representation

Kais Klai, Chiheb Ameur Abid, Jaime Arias, Sami Evangelista

Hybrid Parallel Model Checking of Hybrid LTL on Hybrid State Space Representation

Kais Klai[1], Chiheb Ameur Abid[2,3], Jaime Arias[1], and Sami Evangelista[1]

[1] University of Sorbonne Paris Nord, LIPN, CNRS UMR 7030, Villetaneuse, France
{klai, arias, evangelista}@lipn.univ-paris13.fr
[2] Faculty of Sciences of Tunis, University of Tunis El Manar, 2092, Tunis, Tunisia
[3] Mediatron Lab, SupCom, University of Carthage, Tunisia
chiheb.abid@fst.utm.tn

Abstract. In this paper, we propose a hybrid parallel model checking algorithm for both shared and distributed memory architectures. The model checking is performed simultaneously with a parallel construction of system state space by distributed multi-core machines. The representation of the system's state space is a hybrid graph called Symbolic Observation Graph (SOG), which combines the symbolic representation of its nodes (sets of single states) and the explicit representation of its arcs. The SOG is adapted to allow the preservation of both state and event-based LTL formulae (hybrid LTL), i.e. the atomic propositions involved in the formula to be checked are either state or event-based propositions. We have implemented the proposed model checker within a C++ prototype and compared our preliminary results to the LTSmin model checker.

Keywords: Decision diagrams · Linear temporal logic · Model checking · Parallel verification.

1 Introduction

Model checking [10] has proven to be a major formal verification technique. It is based 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). Thus, model checking is based on an exhaustive exploration of the system state space and, consequently, suffers from the state space explosion problem [31].

The system state space can be represented explicitly (i.e. each state/arc of the graph is represented individually) or symbolically (i.e. the set of the reachable states is represented compactly using decision diagram-based techniques). Hybrid representation of the state space (i.e. an explicit graph where nodes are sets of reachable states encoded symbolically) is also possible, allowing to combine the advantages of both representations. Several approaches

(e.g. [11,19,17,5,30,21,15,6,18,28,23]) 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. In addition to techniques for reduction and compression, parallel and distributed-memory processing can be used [2]. The use of distributed processing increases the speed 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 solutions for both symbolic and explicit model checking (e.g. [2,20,14,4,3]).

A Symbolic Observation Graph (SOG) [18,23] is a graph whose construction is guided by a set of *observable* atomic propositions involved in a formula. These atomic propositions can represent events or actions (event-based SOG [18]), or state-based properties (state-based SOG [23]). 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) [9]. 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 [18,23]. Moreover, once built for a given LTL formula ϕ_1, the SOG can be reused to check any other LTL formula ϕ_2 involving a subset of the atomic propositions of ϕ_1.

In previous works, we have investigated different approaches to parallelize the SOG construction. In [24,25], we propose different algorithms to benefit from additional speedups and performance improvement in execution time and memory saving. However, in some cases where huge state spaces are involved [1], the model checking does not finish due to lack of memory, or it takes too long.

In this work, we present a distributed model checking technique based on the SOG. It extends the multi-core SOG-based model checker introduced in [1] by allowing the handling of huge state spaces. To achieve this, we propose a hybrid technique that combines parallel (shared memory) and distributed (message passing) construction algorithms [27,1]. Roughly, the construction of a SOG is partitioned over a set of processes which, in turn, distribute the building of their sub-graphs over a set of threads. We thus exploit the strengths of the parallel exploration/construction of the SOG, and distribute the processes in charge of the construction and the verification over multiple machines when a single one (although multi-core) is not sufficient.

In the proposed algorithm, both event- and state-based LTL properties can be expressed, combined, and verified. Here, 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. [29,21]), while state-based semantics is more suitable to model-checking approaches [16]. Combining both semantics then allows to express properties in a compact and intuitive manner.

The paper is structured as follows. First, we recall in Section 2 the notions of Kripke structures and hybrid LTL. Then, in Section 3, we introduce the event and state-based SOG. Section 4 describes the main contribution of the paper: a model checker based on the hybrid parallel construction of an event- and state-based SOG. The proposed approach is evaluated and compared to other related works in Section 5. Finally, Section 6 is dedicated to conclusion and perspectives.

2 Preliminaries

In this paper, we consider hybrid linear-time temporal logic (hybrid LTL) formulae where both state- and event-based atomic propositions can occur. Therefore, we chose to represent the semantics (behavior) of a system by a *Labeled Kripke Structure* (*LKS*). Next, we present their formal definition and semantics.

Definition 1 (Labeled Kripke Structure (LKS)). *Let* AP *be a finite set of atomic propositions and* 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*, and*
- $s_0 \in \Gamma$ *is the* initial state*.*

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

- *each member of* $AP \cup Act$ *is a formula,*
- *if* ϕ *and* ψ *are hybrid 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 a hybrid LTL formula is an infinite run $w = s_0 s_1 s_2 \ldots$ (of some *LKS*), assigning to each state s_i a set of atomic propositions and a set of actions that are satisfied within that state. A $p \in AP$ 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, where a single action can occur at a time (i.e. interleaving model of concurrency), at most one action can be assigned to a state of a run.

We write w^i for the suffix of w starting from s_i. Moreover, we say that $p \in s_i$, for $p \in AP \cup Act$, when p is satisfied by s_i. The hybrid LTL semantics is then defined inductively as follows:

- $w \models p$ iff $p \in s_0$, 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 $\exists i \geq 0$ such that $w^i \models \psi$ and $\forall\, 0 \leq j < i$, $w^j \models \phi$.

Thus, an *LKS* K satisfies a hybrid 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), denoted by $LTL \setminus X$, are invariant under the so-called *stuttering equivalence* [10]. Stuttering occurs when the same atomic propositions hold on two or more consecutive states of a given run. In the next section, we will use this equivalence relation to prove that event- and state-based SOGs preserve hybrid $LTL \setminus X$ properties.

3 Event-Based and State-Based SOG

Symbolic Observation Graph (SOG) [23,18] 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 (e.g. LDD [7], a List-implementation of Multiway Decision Diagrams). Despite the exponential theoretical complexity of the size of a SOG (a single state can belong to several aggregates), its size is, in practice, much more reduced than the one of the original reachability graph.

The difference between the event-based and the state-based versions of the SOG ([18] and [23], respectively) is the *aggregation criterion*. In the event-based version, observed atomic propositions correspond to some actions of the system, and aggregates contain states that are connected by unobserved actions. On the other hand, in the state-based version, observed atomic propositions are Boolean state-based conditions, and aggregates regroup states with the same truth values of the observed atomic propositions.

In this section, we present the definition of an **event-state based SOG** which abstracts systems' behavior while preserving hybrid LTL formulae (i.e. both state- and action-based atomic propositions can be used within a same formula). In that sense, the construction of aggregates will depend on both a set of actions and state variables appearing as atomic propositions in the checked formula. Here, systems' behavior will be modeled as Labeled Kripke Structures (LKS).

3.1 Revisiting SOG for hybrid LTL

The adaptation of the SOG to hybrid LTL leads to new aggregation criteria: (1) two states belonging to a same aggregate must have 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 not a member of the same aggregate (even if it has the same label as s), unless it is reachable from another state s'' of the aggregate by an unobserved action.

In the following, we present the formal definition of an aggregate and a SOG, according to a given LKS and the new aggregation criteria discussed above.

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 $Obs \subseteq Act$ be a set of observed actions of $\mathcal{K}$. Then, $UnObs = Act \setminus Obs$ denotes the set of unobserved actions. An aggregate a of $\mathcal{K}$ w.r.t. Obs is a triplet $\langle S, d, l \rangle$ satisfying:*

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

Before defining event-state based SOGs, let us define the following operations:

- $\mathsf{SAT}_{\mathsf{AP}}(S)$: 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:

$$\mathsf{SAT}_{\mathsf{AP}}(S) = \left\{ s'' \in \Gamma \;\middle|\; \begin{array}{l} \exists s \in S, \exists \sigma \in UnObs^*, s \xrightarrow{\sigma} s'' \wedge \\ \forall s' \in \Gamma, \forall \beta \text{ prefix of } \sigma, s \xrightarrow{\beta} s' \Rightarrow L(s) = L(s') \end{array} \right\}$$

- $\mathsf{Out}(a, t)$: returns, for an aggregate $a = \langle S, d, l \rangle$ and action t, the set of states that are reachable from some state in a by firing t. It is defined as follows:

$$\mathsf{Out}(a, t) = \begin{cases} \text{if } t \in Obs & \{s' \in \Gamma \mid \exists s \in S, s \xrightarrow{t} s'\} \\ \text{if } t \in UnObs & \{s' \in \Gamma \mid \exists s \in S, s \xrightarrow{t} s' \wedge L(s) \neq L(s')\} \end{cases}$$

- $\mathsf{Out}_\tau(a)$: returns, for an aggregate a, the set of states whose label is different from the label of any state of a, and which are reachable from some state in a by firing unobserved actions. It is defined as follows:

$$\mathsf{Out}_\tau(a) = \bigcup_{t \in UnObs} Out(a, t)$$

- $\mathsf{Part}_{\mathsf{AP}}(S)$: returns, for a set of states $S \subseteq \Gamma$, the set of subsets of S that defines the smallest partition of S according to the labeling function L. It is defined as follows:
  $\mathsf{Part}_{\mathsf{AP}}(S) = \{S_1, S_2, \ldots, S_n\} \Leftrightarrow S = \bigcup_{i=1}^{n} S_i : \forall i \in \{1..n\}, \forall s, s' \in S_i, L(s) = L(s') \wedge \forall s \in S_i, \forall s' \in S_j, j \neq i, L(s) \neq L(s')$

Now we are able to define the symbolic observation graph for hybrid LTL.

Definition 4 (Event-State Based SOG). *Let $\mathcal{K} = \langle \Gamma, Act, L, \rightarrow, s_0 \rangle$ be an LKS over a set of atomic propositions AP, and $Obs \subseteq Act$ 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 nonempty finite set of aggregates satisfying:*
   - $\forall a \in A,\ \forall t \in Obs, \forall o_i \in \mathsf{Part}_{\mathsf{AP}}(\mathsf{Out}(a,t)), \exists a' \in A$ *s.t.* $a'.S = \mathsf{SAT}_{\mathsf{AP}}(o_i)$
   - $\forall a \in A,\ \forall o_i \in \mathsf{Part}_{\mathsf{AP}}(\mathsf{Out}_\tau(a)), \exists a' \in A$ *s.t.* $a'.S = \mathsf{SAT}_{\mathsf{AP}}(o_i)$
2. *$L' : A \to 2^{AP}$ is a labeling function s.t. $L'(a = \langle S, d, l\rangle) = L(s)$ for $s \in S$;*
3. *$\to' \subseteq A \times Act \times A$ is the action relation where:*
   - $(a,t,a') \in \to' \Leftrightarrow (t \in Obs \wedge \exists o_i \in \mathsf{Part}_{\mathsf{AP}}(\mathsf{Out}(a,t))$ *s.t.* $\mathsf{SAT}_{\mathsf{AP}}(o_i) = a'.S)$
   - $(a,\tau,a') \in \to' \Leftrightarrow (\exists o_i \in \mathsf{Part}_{\mathsf{AP}}(\mathsf{Out}_\tau(a))$ *s.t.* $\mathsf{SAT}_{\mathsf{AP}}(o_i) = a'.S)$
4. *a_0 is the initial aggregate s.t. $a_0 = \mathsf{SAT}_{\mathsf{AP}}(\{s_0\})$*

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 gives to any aggregate of the SOG, the same label as its states. Point 3 defines the action relation: there exists an arc, labeled with an observed action t (resp. τ), from a to a' iff a' is obtained by saturation (using $\mathsf{SAT}_{\mathsf{AP}}$) on a set of equally labeled reached states $\mathsf{Out}(a,t)$ (resp. $\mathsf{Out}_\tau(a)$) by the firing of t (resp. any unobserved action) from states in a. Finally, point 4 characterizes the initial aggregate.

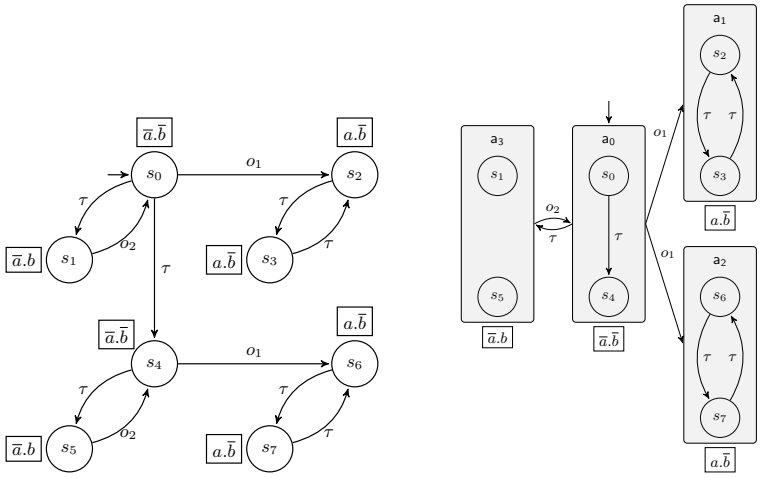
Figure 1(b) illustrates an event-state based SOG corresponding to the LKS of Figure 1(a). This SOG consists of 4 aggregates $\{a_0, a_1, a_2, a_3\}$ and 4 edges. The initial aggregate a_0 is obtained by adding the initial state s_0 of the LKS, and any state labeled similarly to s_0 that is reachable from it by unobserved sequences of actions. Hence, a_0 contains the states s_0 and 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. State s_3 (resp. s_7) is added to a_1 (resp. a_2) because it has the same label as s_2 (resp. s_6) and it is reachable from it by an unobserved action.

According to Definition 4, the SOG associated with an LKS is unique. Thus, by considering the coarsest possible partition of homogeneous successor aggregates, aggregates a_1 and a_2 in Figure 1(b) will be merged into a unique aggregate since they have the same label. Note that SOGs can also be nondeterministic since, for instance, an aggregate can have several successors with τ (i.e. when the reached states, by τ, have different labels).

3.2 Checking stuttering invariant properties on SOGs

The equivalence between checking a given stuttering invariant formula (e.g. $LTL\backslash X$ formula) on the new adapted SOG, and checking it on the original reachability graph is ensured by the preservation of maximal paths (i.e. finite paths leading to a dead state and infinite paths).

Note that a SOG preserves the observed traces of the corresponding model, thus it preserves the infinite runs involving infinitely often observed transitions. Then, the truth value of the state-based atomic propositions occurring in the formulae is visible on the SOG by labeling each aggregate with the atomic propositions labeling of (all) its states. Moreover, the d and l attributes of each aggregate allow detecting deadlocks and livelocks (unobserved cycles), respectively. The detection of the existence of dead states and cycles inside an aggregate is performed using symbolic operations (decision diagram-based operations) only.



(a) Example of LKS

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

Fig. 1. An LKS and its SOG

We can now establish that an *LKS* satisfies a hybrid $LTL \setminus X$ formula iff the corresponding SOG does. The reader can find the proof of Theorem 1 in [27].

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

4 Hybrid LTL model checker based on the SOG

In this section, we propose an on-the-fly parallel/distributed LTL model checking approach based on the event-state based SOG. Here, our aim is to compute the synchronized product between the automaton modeling the negation of the hybrid LTL formula with the SOG (*LKS*), and check its emptiness on-the-fly.

For this purpose, a dedicated process, called *model checker process* is created (see Figure 2). It builds the Büchi automaton of the formula negation, and then it initiates the parallel construction of the SOG simultaneously with the model checking process (i.e. computation of the synchronized product and the emptiness check). Note that this process performs model checking sequentially, while SOG construction is distributed.

Regarding the construction of the SOG, it is performed by running several processes, where each process consists of several threads. The partitioning of the building of the SOG is performed at the process level, where the load balancing between processes is performed statically by using a *hash function*. On the other hand, the partitioning of each part of the graph, for a given process, is performed at the thread level by using a *dynamic load balancing function*.

Since every process has its own memory, during the computation of the synchronized product the model checker process asks *builder processes* for an aggregate: (1) whether it contains a livelock (unobserved cycle) or a deadlock state, and (2) in the memory of which process its successors are stored. The model checker process does not require to receive the LDD structure from builder processes, but rather it receives just a unique identifier for every aggregate. This allows to reduce the size of exchanged messages between the model checker process and the builder ones, as the size of an aggregate can be huge. To do so, we use the hash function MD5 [12], which is also used to decide where an aggregate will be stored during the construction of the SOG. That is, the same function is used to statically balance the load of aggregates construction on builder processes.

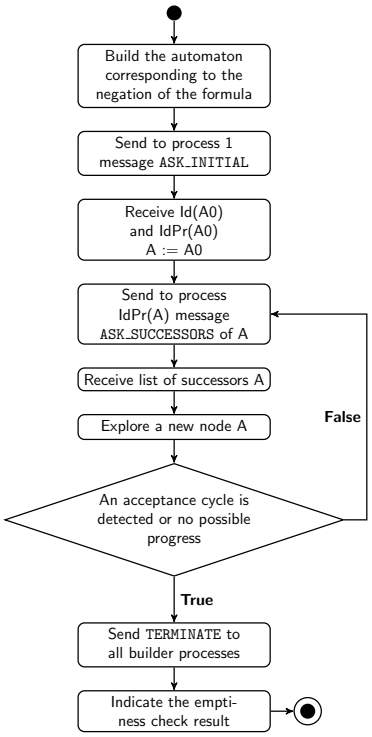
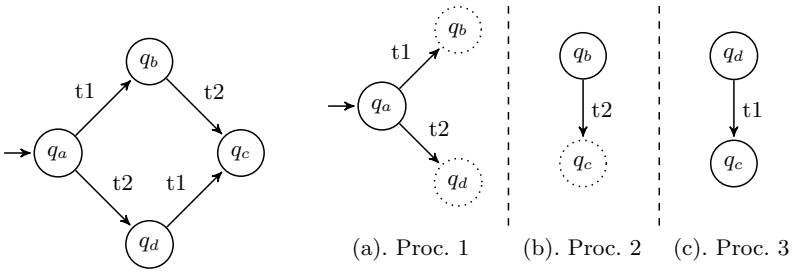


Fig. 2. Algorithm of the model checker process

In order to illustrate how the model checker process retrieves information about a SOG from builder processes, we consider the SOG sample described in Figure 3(a). Let us assume that we have three builder processes, and that the static load balancing produces three graphs as illustrated in Figure 3(b). In this figure, a dotted node of a graph in a process i corresponds to an aggregate such that its LDD structure is not stored by process i. Process i stores only its MD5 value (its unique identifier) and the identity of the process that should store it. For instance, in the graph built by process 1, there is only the aggregate q_a that is stored with its LDD structure by process 1. For the two other aggregates



(a) A SOG sample (b) A distributed SOG

Fig. 3. Illustration of builder processes

q_b and q_d, only their MD5 values are stored. Indeed, q_b is stored in the memory of process 2, while q_d and q_c are stored in the memory of process 3.

With the above in mind, we next describe the algorithm of the model checker process shown in Figure 2. The process starts by requesting from process 1 the initial aggregate. The builder process gets as an answer its MD5 value. When the model checker asks the successors of an aggregate, it sends its request to the process storing the aggregate. For instance, for the SOG of Figure 3(b), when the builder process wants to explore the successors of the initial aggregate, it sends a request to the builder process 1. As an answer to this request, the model checker process should get the MD5 values of aggregates q_b and q_d, and the identities of the processes storing these successors. Following the same approach, in order to get the successors of q_b, the model checker process will send its request to process 2. The termination is fully managed by the model checker process, since it may request information from builder processes even if they terminate the SOG construction. In fact, termination is achieved when the model checker process detects an acceptance run or when it is not possible to progress in the computation of the synchronization product, i.e. the result is an empty set.

Now we describe the algorithm of builder processes (see Figure 4). The process starts by creating several threads. One of them is dedicated for communication with other processes, while the others operate in a loop as follows. At each iteration, it pops a set of markings from which it builds an aggregate. The latter is canonicalized [26] (i.e. aggregates a and a' are equals iff they have the same canonical representation), then through a hash function the thread determines if the computed aggregate should be stored by its process or to be sent to another builder process. In order to minimize communication cost, only markings of the canonicalized version of an aggregate are sent. Function `idPr(A)` returns the process identity that has to store the aggregate `A`. If an aggregate should be sent to another process, only its hash identifier is stored by the current process. Note

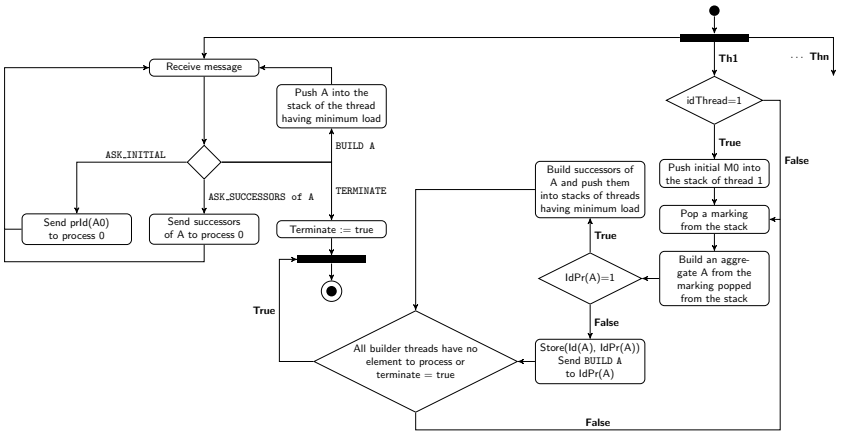


Fig. 4. Algorithm of builder processes

that only thread 1 of the first builder process has to initiate the construction of the distributed SOG by pushing an initial marking into its stack.

Different messages can be received by a builder process from the model checker process. We use the function `Receive(`*TAG*, *message*, *0*`)` to receive a message of type TAG from process with ranking 0. Function `Send(`*TAG*, *message*, *0*`)` allows for sending a message of type TAG to process with ranking 0. A builder process can receive the following types of messages from a model checker process:

- Message `ASK_INITIAL` corresponds to a request from the model checker process to get the MD5 value and the identity of the process storing the aggregate. This message is only received by the master builder process.
- Message `ASK_SUCCESSORS` corresponds to a request from the model checker process to get the list of successors of the aggregate having the MD5 value specified in the message. Also, information about divergence and deadlocks related to this aggregate are concerned.
- Message `TERMINATE` is sent by the model checker process in order to terminate builder processes. This message is sent when the model checker process has already performed the emptiness check.

5 Implementation and Experiments

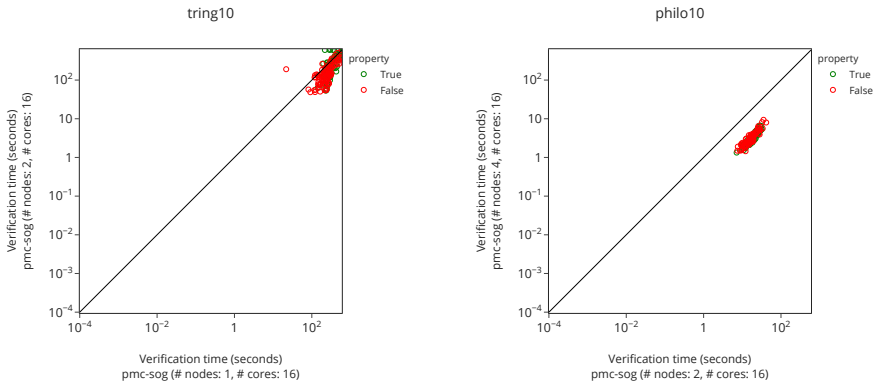
The implementation of the hybrid model checker is based on the Spot library [13]: an object-oriented model checking C++ library that offers a set of building blocks to develop LTL model checkers based on the automata-theoretic approach.

Experiments were performed on the Magi cluster (http://magi.univ-paris13.fr/wiki/) of University Sorbonne Paris Nord and on the Grid'5000 network [8]. For the former, 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. For the latter, we used the `gros` cluster composed of 124 processors connected by a 25 Gbps Ethernet network, and 96GB of RAM.

A total of 5 models from the Model Checking Contest (https://mcc.lip6.fr/models.php) were used in our experiments: *Philosophers* (`philo`), *RobotManipulation*(`robot`), *SwimmingPool* (`spool`), *CircularTrains* (`train`), and *TokenRing* (`tring`). Due to lack of space, we show only some of these figures. The reader can find the files needed to reproduce our experiments and all the figures at https://up13.fr/?G8tMFVS2. Experiments on models `philo` and `train` were performed on Grid'5000, while for the other models they were on Magi.

We measured the time (in seconds) consumed by the verification of 200 random formulae by progressively increasing the number of processes and cores. For this, we set a timeout of 10 minutes. $LTL \setminus X$ formulae were generated by the tool `randltl` (provided by Spot), and filtered into 100 satisfied and 100 violated properties. In the following, the figures are presented using a logarithmic scale, where each point represents a formula (green if satisfied and red if violated).

First, we compared the performance of our hybrid model checker (`pmc-sog`) when using only one process (multi-core execution) and several processes. Figure 5 shows the performance comparison when using 16 cores and 1, 2 and 4



(a) Model `tring10`. Results using 1 and 2 processes with 16 threads each

(b) Model `philo10`. Results using 2 and 4 processes with 16 threads each

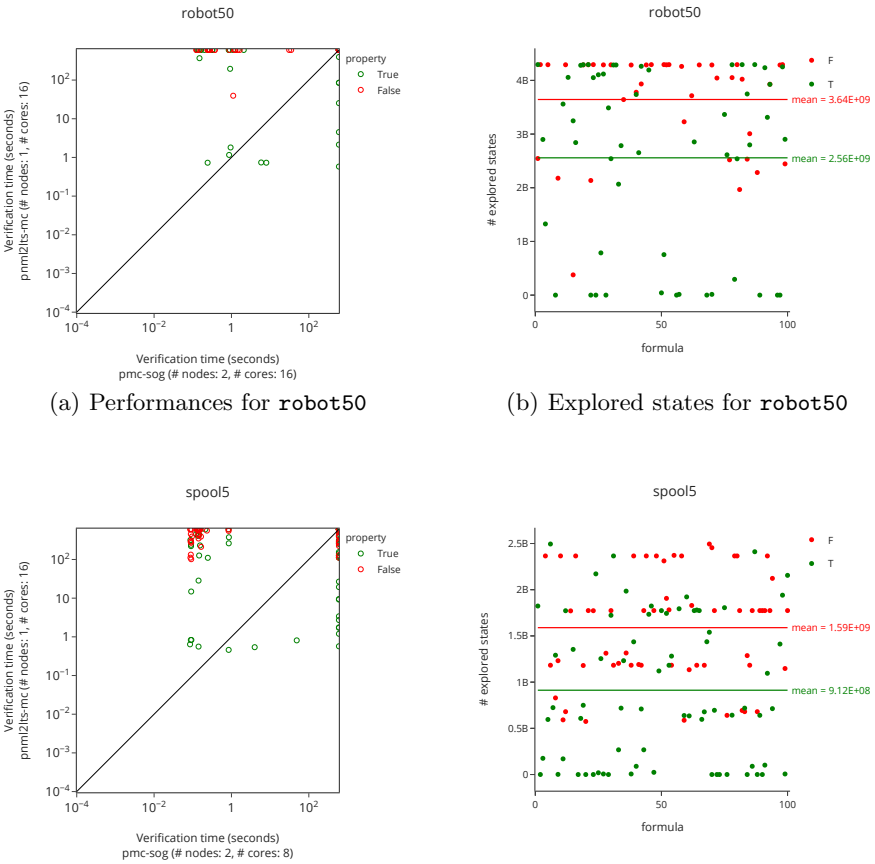
Fig. 5. Comparison of performances in `pmc-sog` by increasing the number of processes

processes with models `tring10` and `philo10` as inputs. Observe that the hybrid execution outperforms the multi-core one, and the verification performance is improved by using more processes despite the induced communication overhead.

Then, we performed a comparison between our tool (`pmc-sog`) and the LTSmin model checker [22]. LTSmin (https://ltsmin.utwente.nl/) is an LTL/CTL/μ-calculus model checker that 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. It is worth noting that we only used formulae whose atomic propositions are based on 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, however it would hide the real performance gains obtained by parallelization. We also avoid resizing of the state storage in all cases by increasing the initial hash table size enough for all benchmarked input models.

We show in Figures 6 and 7 a selection of our experimental results. As we can observe, our approach performs better than LTSmin for big models (i.e. `robot50`, `spool5`). For instance, LTSmin could not verify 96 formulae of `robot50` due to timeout, lack of memory or unknown errors. On the other hand, `pmc-sog` could not verify 49 formulae. We also note that LTSmin has a better performance for small models (i.e. `philo10`, `train12`). Indeed, at the right of each figure the number of states explored by LTSmin during the verification of each formula, confirms this observation. For the model `robot50`, LTSmin explores 3.64×10^9 states on average for verifying violated formulae, and 2.56×10^9 states on average



(a) Performances for `robot50`

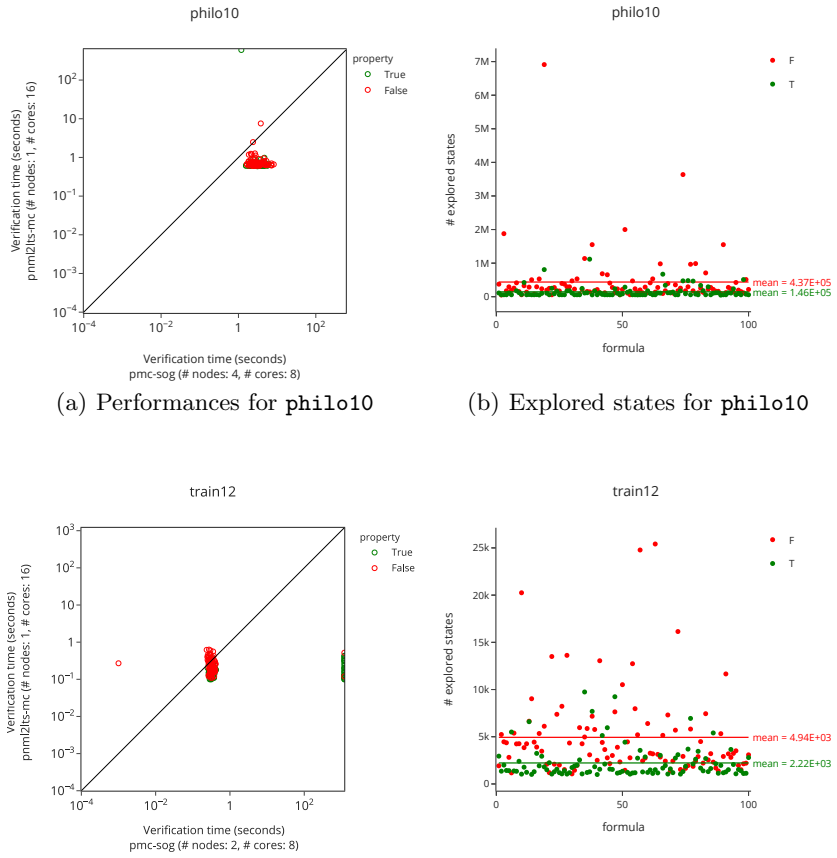
(b) Explored states for `robot50`

(c) Performances for `spool5`

(d) Explored states for `spool5`

Fig. 6. Comparison of `pmc-sog` and `pnml2lts-mc` in big models

for verifying satisfied properties. On the other hand, for the model `train12`, LTSmin explores 4.94×10^3 states on average for verifying violated formulae, and 2.22×10^3 states on average for verifying satisfied properties. This disparity w.r.t. explored states allows LTSmin to quickly verify properties on small models. In this case, the performances of our tool are reduced due to the communication overhead induced by the increase of the number of processes (contrary to LTSmin that does not require any communication since it is a multi-core tool), as well as the time consumed by MD5 computation, the construction and the reduction of the SOG aggregates. Besides, our tool outperforms LTSmin for bigger models since more space memory in total is allocated by the operating system to the tool, and the storage of a SOG is split over processes. Thus, processes can be



(a) Performances for `philo10` (b) Explored states for `philo10`

(c) Performances for `train12` (d) Explored states for `train12`

Fig. 7. Comparison of `pmc-sog` and `pnml2lts-mc` in small models

speedier as they manage less memory space to store parts of the SOG, or they can terminate the model checking while LTSmin cannot due to lack of memory.

To improve our tool performances against relatively small models, we could use some heuristics to stop the local computation of an aggregate, and prioritize the building of the synchronized product and the search for an accepting cycle. A possible heuristic could be a predefined threshold (parameter of our tool) defining the maximum number of states per aggregate. Once this threshold is reached for a current aggregate, its construction is stopped (i.e. split the aggregate). In this case, some aggregates would be connected by a τ transition (i.e. unobserved).

As a preliminary deduction, no model checker has an absolute advantage over the other: our model checker is the fastest for checking properties that require to explore a large number of states, while LTSmin performs better for cases

requiring less states to explore. Finally, it is important to emphasize that our tool is still a prototype and its results should be confirmed by more testing and improving some of its aspects.

6 Conclusion

In this paper, we proposed a hybrid parallel model-checker approach for event- and state-based $LTL\backslash X$ logic. This approach targets both distributed and shared memory architectures, and operates on a hybrid representation of the state space called SOG. Preliminary results of experiments on state-based formulae (only) show that our approach is competitive in comparison with the LTSmin parallel model checker. Our approach has the advantage to handle event- and state-based $LTL \setminus X$, allowing to make formulae containing state and action atomic propositions. This allows to express simple properties intuitively, leading to a smaller state space to explore during the verification process. We plan to pursue the evaluation of our prototype on real word examples and against other model checking tools. Also, many improvements of our algorithm will be investigated, e.g. heuristics to increase efficiency for small models, parallelize the emptiness check, combine symbolic representation and partial order reduction, etc.

References

1. Abid, C.A., Klai, K., Arias, J., Ouni, H.: SOG-based multi-core LTL model checking. In: ISPA/BDCloud/SocialCom/SustainCom. pp. 9–17. IEEE (2020)
2. Barnat, J., Bloemen, V., Duret-Lutz, A., Laarman, A., Petrucci, L., van de Pol, J., Renault, E.: Parallel model checking algorithms for linear-time temporal logic. In: Handbook of Parallel Constraint Reasoning, pp. 457–507. Springer (2018)
3. Barnat, J., Brim, L., Havel, V., Havlícek, J., Kriho, J., Lenco, M., Rockai, P., Still, V., Weiser, J.: DiVinE 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In: CAV. LNCS, vol. 8044, pp. 863–868. Springer (2013)
4. Barnat, J., Brim, L., Rockai, P.: DiVinE 2.0: High-performance model checking. In: HiBi'09. pp. 31–32. IEEE Computer Society Press (2009)
5. Bhat, G., Peled, D.A.: Adding partial orders to linear temporal logic. In: CONCUR. LNCS, vol. 1243, pp. 119–134. Springer (1997)
6. Biere, A., Clarke, E.M., Zhu, Y.: Multiple state and single state tableaux for combining local and global model checking. In: Correct System Design. LNCS, vol. 1710, pp. 163–179. Springer (1999)
7. Blom, S., Van De Pol, J.: Symbolic reachability for process algebras with recursive data types. In: ICTAC. LNCS, vol. 5160, pp. 81–95. Springer (2008)
8. Bolze, R., Cappello, F., Caron, E., Daydé, M.J., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quétier, B., Richard, O., Talbi, E., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. IJHPCA **20**(4), 481–494 (2006)
9. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. **24**(3), 293–318 (1992)
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)

11. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: CAV. LNCS, vol. 531, pp. 233–242. Springer (1990)
12. Dobbertin, H.: Cryptanalysis of MD5 compress. rump session of Eurocrypt **96**, 71–82 (1996)
13. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0 - A framework for LTL and ω-automata manipulation. In: ATVA. LNCS, vol. 9938, pp. 122–129 (2016)
14. Filippidis, I., Holzmann, G.J.: An improvement of the piggyback algorithm for parallel model checking. In: SPIN. pp. 48–57. ACM (2014)
15. Fisler, K., Fraer, R., Kamhi, G., Vardi, M.Y., Yang, Z.: Is there a best symbolic cycle-detection algorithm? In: TACAS. LNCS, vol. 2031, pp. 420–434 (2001)
16. Geldenhuys, J., Valmari, A.: Techniques for smaller intermediary bdds. In: CONCUR. LNCS, vol. 2154, pp. 233–247. Springer (2001)
17. Godefroid, P., Wolper, P.: A partial approach to model checking. In: LICS. pp. 406–415. IEEE Computer Society (1991)
18. 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)
19. Henzinger, T.A., Kupferman, O., Vardi, M.Y.: A space-efficient on-the-fly algorithm for real-time model checking. In: CONCUR. LNCS, vol. 1119, pp. 514–529. Springer (1996)
20. Holzmann, G.J.: Parallelizing the spin model checker. In: SPIN. LNCS, vol. 7385, pp. 155–171. Springer (2012)
21. Kaivola, R., Valmari, A.: The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In: CONCUR. LNCS, vol. 630, pp. 207–221. Springer (1992)
22. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: TACAS. LNCS, vol. 9035, pp. 692–707. Springer (2015)
23. 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)
24. Ouni, H., Klai, K., Abid, C.A., Zouari, B.: A parallel construction of the symbolic observation graph: the basis for efficient model checking of concurrent systems. In: SCSS. EPiC Series in Computing, vol. 45, pp. 107–119. EasyChair (2017)
25. Ouni, H., Klai, K., Abid, C.A., Zouari, B.: Parallel symbolic observation graph. In: ISPA/IUCC. pp. 770–777. IEEE (2017)
26. Ouni, H., Klai, K., Abid, C.A., Zouari, B.: Reducing time and/or memory consumption of the SOG construction in a parallel context. In: ISPA/IUCC/BDCloud/SocialCom/SustainCom. pp. 147–154. IEEE (2018)
27. Ouni, H., Klai, K., Abid, C.A., Zouari, B.: Towards parallel verification of concurrent systems using the symbolic observation graph. In: ACSD. pp. 23–32 (2019)
28. Sebastiani, R., Tonetta, S., Vardi, M.Y.: Symbolic systems, explicit properties: On hybrid approaches for LTL symbolic model checking. In: CAV. LNCS, vol. 3576, pp. 350–363. Springer (2005)
29. Tao, Z., von Bochmann, G., Dssouli, R.: Verification and diagnosis of testing equivalence and reduction relation. In: ICNP. pp. 14–21. IEEE Computer Society (1995)
30. Valmari, A.: A stubborn attack on state explosion. Formal Methods Syst. Des. **1**(4), 297–322 (1992)
31. Valmari, A.: The state explosion problem. In: Petri Nets. LNCS, vol. 1491, pp. 429–528. Springer (1996)