Parallel Model Checking With Lazy Cycle Detection *

Rodrigo T. Saad, Silvano Dal Zilio and Bernard Berthomieu
{rsaad, dalzilio, bernard}@laas.fr

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse France
Univ de Toulouse, LAAS, F-31400 Toulouse, France

Abstract. We propose new algorithms for parallel, exhaustive model checking on multiprocessor architectures. Our approach is designed to emphasize memory efficiency and concurrency and is compatible with common parallel work-sharing policies, such as work-stealing. Moreover, our algorithm makes no particular assumptions about the model or the state class abstractions used during model checking, and therefore it is not restricted to a specific formalism.
Our main contribution is to propose a concurrent, lockless implementation of a semantic model checking algorithm for CTL that does not require to explicitly store the whole transition graph of a model. More precisely, we advocate the use of a *parental graph* data structure, such that we only keep one transition for each state, thus greatly reducing the space complexity of our approach. We show that, in practice, this leads to a good trade-off between execution time and memory consumption. We evaluate the performance of our algorithms on different benchmarks and compare these results with other parallel algorithms proposed in the literature and with existing verification tools.

1 Introduction

Model Checking is a valuable formal verification method that can be used to avoid the presence of logical errors. It has emerged as a promising technique because it offers a "push button" approach to verify finite system. However, there is still a large gap between possible (or decidable) and feasible. Indeed, there are many cases in which it is not possible to perform the verification of a finite system due to the *state explosion* problem. That is, the number of states that should be inspected can grow exponentially larger in function of the complexity of the system. The state explosion problem is one of the main challenges faced by model checking researchers.

In this work, the idea is to take benefit of recent advances on the hardware side to improve enumerative model checking techniques face the state explosion problem. Indeed, we now have access to computers with larger shared memory

* This work was partially supported by the JU Artemisia project CESAR, the AESE project Topcased and the Région Midi-Pyrénées

space and the multi-core architectures that makes feasible the verification of larger models, in a reasonable amount of time.

We describe and analyze a new parallel model checking approach for shared memory architecture that is "compatible" with the parallel state space generation techniques we presented in [15]. By compatible, we mean that we base our approach on the same set of hypotheses; actually, we should say the same absence of restrictions. We assume that we are in the least favorable case, where *we have no restrictions on the models that can be analyzed*, and the algorithm should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing.

We decided to define our own parallel algorithm for model-checking instead of trying to parallelize existing, state-of-the-art, sequential algorithms. We can give a simple, theoretical justification to support our choice. In the sequential case, many efficient model-checking algorithms rely on the computation of Strongly Connected Components (SCC) or, at least, rely on following a specific order when exploring the state space graph—generally a Depth-First Search (DFS) or breadth-first search order. This is the case, for example, in most of the *automata-theoretic* approaches for model-checking Linear Temporal Logics (LTL). These algorithms rely on efficient methods to detect the presence of cycles in a graph, such as Tarjan's algorithm [16] or "nested-DFS" [6]. While this class of sequential algorithms are very efficient—their complexity is linear on the size of the state graph—they do not lend themselves to parallelization [13].

We can give a second justification, these algorithms are appreciated because they are able to find an error before the complete construction of the state space. However, the parallel implementation of these so called on-the-fly algorithms fails to deliver the same efficiency for the case when the formula is valid, i.e. the system respects the property.

Based on these observations, we decided to develop a new algorithm for parallel model checking oriented to the case when the formula is valid, trading the extra computing power for a better memory efficiency. We follow an alternative approach that we call *semantic model-checking*. This is the approach initially proposed by Clarke and Emerson [5] for Computation Tree Logics (CTL) model-checking. In its simplest form, a semantic algorithm works by labeling each state of the system with the "sub-formulas" of the initial specification that are true for this given state. Labels are computed iteratively until we reach a fix-point, that is until we cannot add new labels.

Our approach is quite simple. The algorithm is based on two separate steps: (1) a forward, constrained exploration of the state graph—where we start labeling each state with local information—followed by (2) a backward traversal—where we propagate information towards the root of the state graph—to check if the resulting graph has an infinite path.

We propose two versions of our algorithm that differ by the way we store the state graph in memory. In the first version, we assume that, for every reachable state, we have a constant time access to the list of all its "parents". Basically, it means that we store the *reverse graph* (RG) structure of the state space. In

the second version, we assume that we have access to only one of the parents, meaning that we may have to recompute some transitions dynamically. We say that the second version is based on a *reverse parental graph* (RPG).

The advantage of this second version is to save memory space. The gain in memory space can be very important; something we have experienced in our experiments. Indeed, if we use the symbol S to denote the number of states (vertices) in the graph, the size of the data structure for our first algorithm is of the order of $O(S^2)$, in the worst case, while it is of the order of $O(S)$ for the second version.

This work is organized as follows. Section 2 presents the related work and the contributions of this work. After the definition of basic results on graph theory in Section 3, we describe our parallel algorithms using pseudo-code in Section 4. Before concluding in Section 7, we give a set of experimental results in Section 5 and a benchmark comparison with the tool Divine in Section 6.

2 Related Work

The formal verification of behavioral properties stumbles into algorithms for cycle detection. The efficiency of these algorithms are mandatory for the overall performance. This problem has been addressed by the sequential model checkers through the use of the well known Tarjan [16] and Nested-DFS [6] algorithms. The sequential automata-theoretic approaches based on these algorithms are able to deliver efficient on-the-fly solutions both in terms of time and space. However, the same can not be said for the solutions proposed for parallel shared memory machines.

It is known since the 1980's that "finding a cycle in a graph is an inherently sequential problem" [13], more precisely, that it is related to problems that are P-space complete (see problem A.2.18 in [7]). This gives strong evidence that trying to parallelize this class of automata-theoretic, sequential, algorithms is not the right way to go, at least if we expect a significant speedup. (and assuming we use a polynomial number of processors).

The search for an efficient parallel on-the-fly algorithm is one of the most activity research branch in parallel model checking. The researchers who have accepted this challenge have chosen to fallow the Automata-Theoretic approach for LTL model checking. The difficulty in to reuse the sequential cycle detection algorithms (Tarjan or Nested-DFS) in order to determine whether an accepting state is part of a cycle. Two works stand out, which one implemented in the context of a tool. We have Divine with the **owcty + map** algorithm [1] and LTSmin with the **mc-ndfs** algorithm [10]. They differ basically by the algorithm used to detect cycles.

The **owcty + map** is the state of the art algorithm for parallel LTL model checking; it was presented by Barnat et al. by combining the **owcty** and **map** algorithms. The alliance of these two techniques resulted in "a parallel on-the-fly linear algorithm for LTL model checking of weak LTL properties". (Weak LTL properties are those expressible by an automata that has no cycle with

both accepting and no-accepting states on its path.) However, the algorithm complexity may be quadratic if the LTL property does not meet this requirement.

More recently, the distributed algorithm **swarm** [8] had been extended for multi-core architectures in [10] and named *multi-core nested DFS* (**mc-ndfs**). They proposed a multi-core version with the distinction that the storage state space is shared among all workers in conjunction with some synchronization mechanisms for the nested search. Even if in the worst-case each processor might still traverse the whole graph , i.e. unable to scale in the worst case, this work goes one step further to propose an on-the-fly algorithm because the time complexity is still linear in the size of the graph.

In contrast with the number of solutions proposed for parallel LTL model checking, just two were specifically conceived for parallel CTL model checking. We have [9] that supports CTL* and [12] that supports μ-calculus (subsumes CTL).

2.1 Contribution

Our algorithms follow the classical semantic approach proposed by Clark et al. in [4], with the distinction that we only support a subset of CTL formulas. We follow an approach based on labeling states, similar to the one used in [2] and [3] for game automaton on distributed memory machines. We chose a semantic approach because we believe that it is more appropriate for a parallel algorithm with dynamic work-load strategies.

We define a new algorithm, that we call MCLCD for Model Checking Algorithm with Lazy Cycle Detection. We propose it in two versions: a first version based on a reverse traversal of the state graph, called RG, where we need to explicitly store the transitions of the system; and a second version, RPG, where we only need to store a spanning subgraph. The RG version has a linear time and space complexity, in $O(|S| + |R|)$, while the RPG version has a time complexity in $O(|S| \cdot (|R| - |S|))$ and a space complexity in $O(|S|)$.

Our main contribution, from the algorithmic viewpoint, is the definition of an algorithm based on the reverse parental graph (RPG). To the best of our knowledge, this approach is totally new. Indeed, most model-checking algorithms for CTL avoid to store the transition relation explicitly. But these approaches always rely on some assumptions about the models, for instance that it is possible to compute the "reverse" transition relation efficiently. (This is the case, for example, when model-checking Timed Petri Nets [11] using State Class Graphs.) We do not make this assumption in our case (this assumption is not valid, for example, with models that mix real-time constraints and data variables). We still define a version of our algorithm based on the reverse transition graph because it is useful to prove the soundness of our method and for studying the theoretical complexity.

We build our algorithms over the parallel state space explorer we presented in [15]. It is based on a mixed design that enables the use of distributed hash tables as a single shared concurrent has map. Roughly speaking, the global state space is stored in a set of local hash tables, each controlled by a different processor,

while only a small part of the shared-memory is used for coordinating the state space exploration.

Finally, we make use of the work-stealing strategy to share work among the processors for both the state space construction phase and the property validation (cycle detection) phase. Related works [10,9] use dynamic workload policies for the parallel state space construction only, they do not employ any kind of work-load approach during cycle detection: Inggs et al. perform (independent) local cycle detection procedures whenever a node was revisited; and Laarman et al. propose an on-the-fly algorithm where each process performs its own nested search and shares information only to avoid the repetition of nested searches.

3 Some Graph Theoretical Properties

In this section we present a summary of the definitions and theorems we use in our algorithm. A complete study, together with their respective proofs, is presented in [14].

Our model-checking algorithm is based on an iterative exploration of the state space graph. To this end, we need to define some properties of Directed Acyclic Graphs (DAG).

Definition 1. *A finite Directed Graph $G(V, E)$ is an ordered pair (V, E) comprising a finite set V of vertices and a finite set E of edges, (v_i, v_j), such that v_i and v_j are in V for all edges. A finite Directed Acyclic Graph (DAG) is a finite directed graph $G(V, E)$ with no* cycles*, that is there is no way to find a sequence of edges $v_0 \cdot \ldots \cdot v_{n+1}$ such that $(v_i, v_{i+1}) \in E$ for all index i in $0..n$ and $v_0 = v_{n+1}$.*

We prove in [14] that, in a finite DAG, there is always at least one vertex that has no children (what we call a *leaf*) and one vertex without parents (what we call a *root*). In the following, we say that a leaf has *out-degree zero* and that a root has *in-degree zero*.

Lemma 1. *In a finite DAG $G(V, E)$ there exists at least one vertex in V with in-degree zero and at least one vertex in V with out-degree zero.*

We give another property related to leaves in a DAG. Our algorithm mostly relies on the following observation: a finite graph is acyclic if, whenever we recursively remove all the leaves, we eventually ends up with an empty graph. By recursively removing the leaves, we mean removing a leaf from the graph—together with all its incoming edges—and starting over with the remaining graph. The procedure stops when no more nodes can be removed.

Actually, we use a slightly stronger property and rely on the fact that it is enough to stop removing leaves when all the vertices have in-degree zero (the graph has only root nodes). This property is expressed by Theorem 1.

Theorem 1. *A finite directed graph $G(V, E)$ is a DAG if and only if, by recursively removing the leaves, we finally end up with a graph that only has root nodes.*

The complete proof for Theorem 1 is presented in [14]. To conclude this section, we show some properties of Parental Graphs, that is a spanning subgraph such that all the nodes, except the root(s), have an in-degree of one.

Definition 2. *We say that a directed graph, $PG(V_p, E_p)$, is a parental graph of $G(V, E)$ if: (1) PG is a subgraph of G that has the same vertex set (that is $V_p = V$ and $E_p \subseteq E$) and (2) for every vertex $v \in V$, if v is not the root in G then v has an in-degree of one in PG.*

To obtain a parental graph PG, from a directed graph G, it is enough to keep only one edge coming in for every vertex in G and delete the others. Also, if G is acyclic, then all its parental graphs are acyclic.

The following theorem states an important connection between a graph and its parental graphs: if PG is a parental graph of (the finite directed graph) G, then the set of leaves of PG subsumes the leaves of G. Indeed, a leaf of G is necessarily a leaf of PG, but the opposite may be false. Thus, we can also conclude that G has necessarily some cycles if we fail to find an out-degree zero vertex in PG that is also in G.

In our algorithm, we use the leaves of a parental graph PG as a set of candidates—an approximation—for finding the leaves of G, saving us from testing all the nodes in the graph.

Theorem 2. *Let G be a finite directed graph and PG be a parental graph of G. If the graph G is acyclic then PG has at least one leaf that is also a leaf in G.*

Proof. Let G be a finite directed graph and PG be a parental graph of G. By Lemma 1, since G is acyclic, there is at least one leaf in G; Moreover, since PG is a subgraph of G, a vertex of out-degree zero in G must also have out-degree zero in PG (a parental graph has less edges). Therefore the leaf in G is also one of the leaf of PG.

In this work, we will essentially work with *reverse graphs* and *reverse parental graphs*.

Definition 3. *The reverse graph of a directed graph $G(V, E)$ is the graph $G^{-1}(V, E^{-1})$ such that the edge (u, v) is in G if and only if the edge (v, u) is in G^{-1}. A reverse parental graph of G is a parental graph of G^{-1}.*

4 A Model Checking Algorithm with Lazy Cycle Detection

Our parallel algorithms for model checking, named MCLCD (for Model Checking With Lazy Circle Detection), is based on two separate steps: (1) a forward exploration of the state graph (in collaboration with the state space construction), where we label each state with some "local" information; followed by (2) a backward traversal—and label propagation phase—to check if the resulting graph is a DAG.

In the second step, we do not explicitly look for cycles (like in a "nested-DFS" approach for example). We rather follow a "lazy approach" in order to avoid all the inherent complexities related to the parallel detection of cycles. The second step can be easily implemented in parallel, each processing unit updating the labels of its own states.

A first optimization is to constraint the state space exploration in order to generate only the portion of the state graph that is important to prove or disprove the specification. This approach is quite similar to techniques for on-the-fly model-checking because, for some class of formulas, we can sometimes disprove the specification before generating the complete state space. For example, in the case of reachability formulas, such as the invariant formula $A\Box(\phi)$, we will of course stop exploring as soon as we find a state satisfying the predicate $\neg\phi$.

The backward traversal is performed only for safety and liveness formulas; it is not necessary for reachability formulas. We define these different classes of formulas in Figure 1 and list, for each formula, whether they involve a backward step. Concerning the set of supported formulas, our specification language includes formulas for expressing basic reachability, safety and liveness formulas and can be expressed as a subset of CTL formulas, with the distinction that we follow a *local*, instead of global, model checking semantic.

Formula	Interpretation	Forward	Backward	Classification
$E\ (\psi \cup \phi)$	$E\ (\psi \cup \phi)$	x		Reachability
$A\ (\psi \cup \phi)$	$A\ (\psi \cup \phi)$	x	x	Liveness
$E\Diamond(\phi)$	$E\ (\text{True} \cup \phi)$	x		Reachability
$A\Diamond(\phi)$	$A\ (\text{True} \cup \phi)$	x	x	Liveness
$E\Box(\phi)$	$\neg A\Diamond(\neg\phi)$	x	x	Safety
$A\Box(\phi)$	$\neg E\Diamond(\neg\phi)$	x		Safety
$\psi \rightsquigarrow \phi$	$A\Box(\neg\psi \vee A\Diamond\phi)$	x	x	Liveness
$A\Box A\Diamond(\phi)$	$true \rightsquigarrow \phi$	x	x	Liveness

Fig. 1. List of Supported Formulas.

From the interpretation of formulas listed in figure 1, we see that it is enough to provide a model-checking procedure for only three formulas: (reachability) $E\,(\psi \cup \phi)$, (liveness) $A\,(\psi \cup \phi)$, and (leadsto) $\psi \rightsquigarrow \phi$. We describe our model-checking procedure for each of these three cases.

4.1 Notations

We assume that we perform model-checking on a Kripke System $KS(S, R, s_0)$. We will use, interchangeably, the notation KS for the Kripke structure (S, R, s_0) and G for the directed graph $G(S, R)$, also called the state graph.

The expression $|S|$ is used to denote the cardinality of S (and therefore the number of reachable states), while $|R|$ is the number of transitions. Inside asymptotic notations (big O notations) we will simple use the symbols S and R when we really means $|S|$ and $|R|$.

We assume that every state $s \in S$ is labeled with a value, denoted $\text{suc}(s)$, that record the out-degree of s in KS. The value of $\text{suc}(s)$ is set during the forward

exploration phase. Initially, $\mathrm{suc}(s)$ is the cardinality of the set of successors of s in KS, that is $\mathrm{suc}(s) = |\{s' \mid s\, R\, s'\}|$. We decrement this label during the backward traversal of the state graph; when the value of $\mathrm{suc}(s)$ reaches zero, we say that s is *cleared* from the state graph. In our pseudo-code, we use the expression $\mathrm{suc}(s)$.dec() to decrement the value of the label suc for the state s in KS, and the expression $\mathrm{suc}(s)$.set(i) to set the label of s to some integer value i.

When we deal with the reverse parental graph version of our algorithm, we assume that we implicitly work with one particular parental graph of KS, denoted PKS. In this case, we assume that every state $s \in S$ is also labeled with a value, denoted $\mathrm{sons}(s)$, that record the out-degree of s in KS. We also label each state $s \in S$ with a state, denoted $\mathrm{father}(s)$, that is the predecessor of s in PKS. (The label $\mathrm{father}(s)$ makes sense only if s is not s_0, the initial state of KS.)

Initially, the value of $\mathrm{sons}(s)$ is equal to zero. The value of this label will be incremented during the forward exploration of KS, when we build PKS (that is, we select the transitions from KS that will be stored in PKS). This operation is denoted $\mathrm{sons}(s)$.inc() in our pseudo-code. We will decrement the value of $\mathrm{sons}(s)$ during the backward traversal phase.

4.2 Model-checking Reachability properties — $\boldsymbol{E\,(\psi \cup \phi)}$

To check the formula $E\,(\psi \cup \phi)$, we basically search for states satisfying the predicate ϕ in the state graph. More precisely, we stop exploring a path whenever we find a state such that (1) ϕ holds or (2) $\neg\psi \wedge \neg\phi$ holds. In the first case, we can stop the exploration and return that the property is true. Otherwise, we stop the exploration on this path because the property does not hold. The exploration continues over the set of unexplored paths until (1) or (2) holds. The property is false if (1) never holds.

The function is the same for the two versions of our algorithm; based on the reverse graph or the reverse parental graph data structure.

4.3 Model-Checking Liveness Properties — $\boldsymbol{A\,(\psi \cup \phi)}$

To check the formula $A\,(\psi \cup \phi)$, we basically search for states satisfying the predicate ϕ in the state graph. Like for reachability properties, we stop exploring a path whenever we find a state such that (1) ϕ holds or (2) $\neg\psi \wedge \neg\phi$ holds.

If we find an occurrence of case (2), we now at once that the property is false. In the other case, we start a second phase, after the forward exploration is over, in order to detect cycles. We call this second phase the *clearing phase*, because it consists in recursively removing the leaves node from the graph. This process ends either when we finally reach the initial state (which mean the property is true), or when no states with zero out-degree can be found (in which case we know that there is a cycle). The validity of this process is a direct corollary of Theorem 1.

```
1  function BOOL check_a(ψ : pred, φ : pred, s0 : state)
2     Stack A ← new Stack(∅) ;
3     // Start with the forward exploration
4     if forward_check_a(ψ , φ, s0, A) then
5        // If all forward constraints are respected, start the backward phase
6        return backward_check_a(s0, A)
7     else
8        // We found a problem during the forward exploration
9        return FALSE
10    endif
```

Listing 1.1. Algorithm for the formula $A(\psi \cup \phi)$

We give the pseudo-code for checking the formula $A(\psi \cup \phi)$ in Listing 1.1. The inputs are the atomic properties ψ and ϕ and the initial state s_0. The algorithm uses a stack, A, to collect the states where ϕ holds during the forward exploration phase. The algorithm also uses two auxiliary functions, forward_check_a and backward_check_a, that are defined later. The implementation of these two functions depend on the version of the algorithm that we use. We start by studying the case where we use the *Reverse Graph* data structure and then the *Reverse Parental Graph*.

Algorithm for the reverse graph version — RG

We give the pseudo-code for the function forward_check_a in Listing 1.2. The last parameter of this function, A, is a stack that is used to collect the "leave nodes" of the state graph; the state where ϕ holds. These states will be the starting points in our backward traversal of the graph.

During the forward exploration phase, we label each state s with a value that is the number of successors of s in the initial state graph (the Kripke structure). During the backward traversal phase of the algorithm, we will decrement this value each time we remove a successor of s. Intuitively, a state can be removed as soon as it is tagged with zero. We never actually remove a state from the graph. Instead, when a processor change the label of a state s to 0[1], we also decrement the labels of all the parent of s in the graph. Hence the choice of storing the reverse of the transition function in the data structure.

Listing 1.2 gives the pseudo-code for the function backward_check_a, that implements the clearing phase. We start by clearing all the states in A which are, by construction, states s such that suc(s) is zero. When a state is cleared, we decrement the label of all its parents ($suc(s').dec()$) and check which ones can be cleared ($suc(s') == 0$). The algorithm stops if the initial state, s_0, can be cleared or if there are no more state to update.

Algorithm for the reverse parental graph version — RPG

The forward exploration function, forward_check_a, for the parental graph version is similar to the one presented for the reverse graph (see Listing 1.2). The

[1] We assume that the decrementing operations are done in parallel.

difference, in this case, is that we only have access to the reverse parental graph data structure. As a consequence, we can only access one of the parents of a state in constant time (what we call the father of the state). In our implementation, we choose has father for a state s', the first state, say s, that leads to s' in the exploration. From line 15 of Listing 1.2, the RPG version of forward_check_a executes two additional statements. It takes into account the father reference s for each state s' ($father(s').set(s)$) and increments the number of sons of s ($sons(s).inc()$). These information help the backward traversal to track non cleared leaves.

For the clearing phase, we rely on the parental graph structure to "propagate" the cleared states toward the root of the state graph. We give the pseudo-code for the backward traversal phase in Listing 1.3. The algorithm iterates between two behaviors, *clearing* and *collecting*. The *clearing* behavior is similar to the pseudo-code for the RG algorithm (see Listing 1.2), with the difference that we decrement only the father of a state and not all the predecessors. When there is no more labels to decrement—and if the initial root is not yet cleared—the algorithm starts looking for states that can be cleared. For this, we test all the states s such that $sons(s) == 0$; that is, such that all the sons of s have been cleared. In this case, to check if s can be cleared, we have to recompute all its successors in KS (because this information is not stored in the RPG) and check whether they have been cleared also (if their suc label is zero).

The advantage of this strategy is that we do not have to consider all the states in the graph, just a subset of it. Indeed, we know from Theorem 2 that this subset is enough to test the presence of a cycle. At the opposite, the drawback of this approach is that we may try to clear the same vertex several times, which may be time consuming.

4.4 Model-Checking the Leadsto Property — $\psi \rightsquigarrow \phi$

To check the formula $\psi \rightsquigarrow \phi$, we need to prove that there is no cycle that can be reached from a state where ψ holds, without first reaching a state where ϕ holds. Indeed, otherwise, we can find an infinite path where ϕ never holds after an occurrence of ψ. Figure 2 gives an example of graph for which the formula is valid.

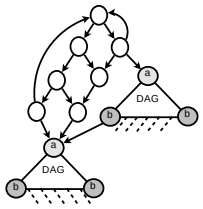


Fig. 2. Leadsto $a \rightsquigarrow b$ where a is ψ and b is ϕ.

This observation underlines the link between checking the formula $\psi \rightsquigarrow \phi$—locally, for the initial state—and checking the validity of $A\Diamond(\phi)$—globally, at every state where ψ holds. As a consequence, we can use an approach similar to the one used for liveness properties in the previous section. The main difference is that, instead of clearing the initial state, we have to clear all the states where ψ holds.

We skip the presentation of RG and RPG pseudo-code because they are similar to the ones presented for the formula $A(\psi \cup \phi)$. The main difference is

```
1  function BOOL forward_check_a(ψ : pred, φ : pred, s_0 : state, A : Stack)
2     Set S ← new Set(s_0) ;
3     Stack W ← new Stack(s_0) ;
4     while (W is not empty) do
5        s ← W.pop();
6        if (s ⊨ φ) then
7           suc(s).set(0) ; // we clear state s from KS
8           A.push(s)
9        elsif (s ⊨ ψ) then // we tag s with its number of successors
10          suc(s).set(number of successors of s in KS) ;
11          if (suc(s) = 0) // check if s is not a dead state
12             return FALSE
13          forall s' successor of s in KS do // and continue the exploration
14             if (s' ∉ S) then
15                S ← S ∪ {s'} ; // s' is a new state
16                W.push(s')
17       else return FALSE
18    return TRUE
19
20 function BOOL backward_check_a(s_0 : state, A : Stack)
21    while (A is not empty) do
22       s ← A.pop() ;
23       if (s = s_0) then // the property is true if
24          return TRUE     // we reach the initial state
25       forall s' parent of s in KS do // otherwise we check if the
26          suc(s').dec() ;               // predecessors of s can be cleared
27          if (suc(s') = 0) then
28             A.push(s')
29    return FALSE
```

Listing 1.2. Forward and backward exploration for $A(\psi \cup \phi)$ with Reverse Graph

```
1  function BOOL backward_check_a(ψ : pred, φ : pred, s_0 : state, A : Stack)
2     over ← FALSE
3     while (not over)
4        while (A is not empty) do
5           //Clearing
6           s ← A.pop() ;
7           if (s = s_0) then // the property is true if
8              return TRUE       // we reach the initial state
9           s' ← father(s) ; // otherwise we check if
10          sons(s').dec() ; // the father of s can be cleared
11          suc(s').dec() ;
12          if (suc(s') = 0) then
13             A.push(s')
14       //Collecting: if we have no more states to clear in A we try to find
15       // candidates among the states with no children in PKS
16       forall s such that sons(s) = 0 and suc(s) ≠ 0 in KS do
17          if test(s) then
18             suc(s).set(0) ;
19             A.push(s)
20       if (A is empty) then
21          over ← TRUE //No good candidate was found, end backward search
22    return FALSE
23
24 function BOOL test(s : state)
25    forall s' successor of s in KS do
26       if suc(s') ≠ 0 then
27          return FALSE // at least one successor is not cleared
28    return TRUE
```

Listing 1.3. Backward exploration for $A(\psi \cup \phi)$ with Reverse Parent Graph

in the termination condition: the function returns true if all the states where ψ holds are labeled as cleared.

4.5 Correctness and Complexity

The correctness proof (Termination, Completeness and Soundness) and the complexity study of our algorithm is presented in details in [14]. We skip its presentation here due to the lack of space. Below, we show the worst-case complexity of our algorithm in the sequential case. We more specifically present the case of the liveness formula $A(\psi \cup \phi)$. The results for this case can be generalized to our whole logic. (Recall that, inside asymptotic notations, we use the symbols S and R when we really means $|S|$ and $|R|$.)

Complexity: The worst-case time complexity of the algorithm is in the order of $O(S + R)$ for the RG version and in the order of $O(S \cdot (R - S))$ for the RPG version.

Since the number of transitions in a graph $G(S, R)$ is bounded by $|S|^2$, we obtain a complexity in the order of $O(S^2)$ for the RG version and of $O(S^3)$ for the RPG version.

In conclusion, we want to emphasize that the RG version of the algorithm has a time complexity that is a factor of $|S|$ better than the RPG version. At the same time, the space complexity is better for the RPG version than for the RG version by the same factor: the space complexity is in $O(S)$ for the RPG version and in $O(S^2)$ (or $O(S + R)$) for the RG version.

4.6 Parallel Implementation of our Algorithm

While we presented our algorithm, we have not specifically fixed the abstract computational model that is used to interpret the semantics of our pseudo-code. We can easily adapt the same code to a Parallel RAM model, following the Single Program Multiple Data (SPMD) programming style that we adopted for our algorithms presented in a previous work [15].

In a SPMD context, all processing units will execute the same functions, as defined in Sect. 4. Following this approach, the forward exploration phase and the cycle detection (backward traversal) phase can both be easily parallelized. Then, for the model-checking function themselves—for instance the function check_a—we only need to synchronize the termination of the forward exploration with the start of the backward label propagation. At each point, a processing unit can terminate the model-checking process if he can prove (or disprove) the validity of the formula before the end of the exploration phase.

We consider a shared memory architecture where all processing units will share the state space (using the mixed approach that we presented in [15]) and where the working stacks are partially distributed (such as the stacks W, A and P used in our pseudo-code). For most of our pseudo-code, it is enough to rely on atomic compare and swap primitives to protect from parallel data races

and other synchronization issues; typically, compare-and-swap primitives will be used when we need to test the value of a label or when we need to update the label of a state (for instance with expressions like sons(s).dec()). Together with the compare-and-swap primitive, we use our combination of distributed, local hash tables with a concurrent localization table to store and manage the state space.

For the RG version of the algorithm, we can ensure the consistency of our algorithm by protecting all the operations that manipulate a state label. (We made sure, in our pseudo-code, that every operation only affect one state at a time.)

The parallel version of RPG is a bit more complicated. This problem is related to the behavior *collection*, that needs to check all the successors of a given state to see if they are cleared. First, this operation is not atomic and it is not practical to put it inside a critical section (it would require a mutex for every state). If two processors collects the same state, then the father of this state will be decremented two times (later) at the *clearing* procedure. Second, the *collection* operation must be performed after all processors have finished the *clearing* operation, otherwise, Theorem 2 can not be applied to our algorithm. For instance, if the processors are allowed to perform asynchronously both *clearing* and *collection* operations, then a state may be forgotten to be collected because one of its successors has not been cleared yet.

We solve the parallel issues for RPG through the synchronization of all processors before both *clearing* and *collection* operations. The synchronization ensures that no states will be forgotten to be collected. Then, we take advantage of our distributed local hash tables to avoid the concurrent access problem. Each processor is restricted to perform the *collection* operation over the states stored in its own table.

To conclude with the parallel version of our algorithm, we use a work-stealing strategy (see [15]) to balance the work-load between the different phases of our algorithm. During the exploration phase, we use the same strategy than in our algorithm for parallel state space construction, where each processor holds two stacks for unexplored states (one private stack and one shared stack). For the backward traversal, we use the same idea of two stacks for the accepted vertices (the stack called A in our pseudo-code); whenever a thread has no more vertex to clear, it tries to "steal" non-cleared vertices from other processors.

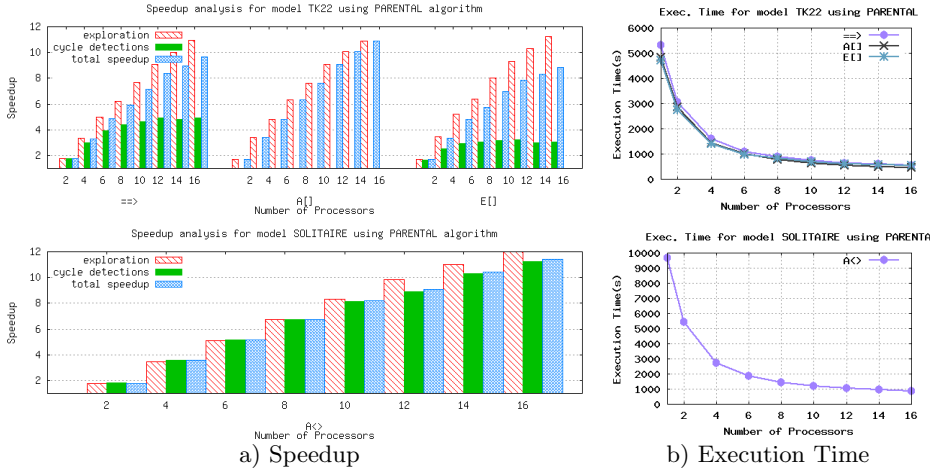
5 Experimental Results

In this section we present a summary of the experimental results we obtained with our prototype implementation of MCLCD. The complete set of experiments is presented in [14]. We have implemented three versions of our model-checking algorithm as part of our prototype model checker MERCURY ([14]). They are built on top of our previous algorithm for parallel state space exploration (see [15]). Experimental results presented in this section were obtained on a Sun Fire x4600 M2 Server, configured with 8 dual core opteron processors and 208 GB of RAM memory, running the Solaris 10 operating system.

For this benchmark, we use a set of 8 classical models— Token Ring protocol (TK), Peg-Solitaire (Peg), etc— with a mix of valid and invalid properties. We experimented with all the formulas: reachability ($E\Diamond\phi$), safety ($A\Box\phi$ and $E\Box\phi$), liveness ($A\Diamond\phi$) and leadsto ($\psi \rightsquigarrow \phi$). (See [14] for a complete description of the models and formula used in this benchmark.)

5.1 Speedup Analysis

In this section, we study the relative speedup and the execution time for our algorithms. In addition, we also give the separate speedup obtained in each phase of the algorithm—during the exploration (forward) and cycle detection (backward) phases—in order to better analyze the advantages of our approach.



a) Speedup b) Execution Time

Fig. 3. Speedup and execution time analysis for Token Ring and Solitaire models.

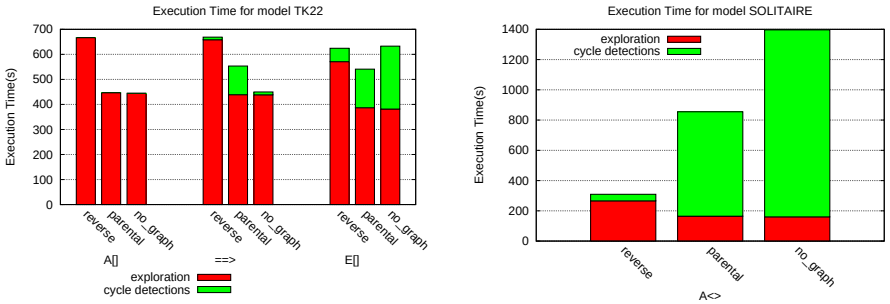


Fig. 4. Comparison with a Standard Algorithm.

Figure 3 presents a speedup analysis for the RPG (parental) version of our algorithms. We present only two models— Token Ring (TK) and Solitaire game with 33 pegs —because they are enough to summarize the results we obtained in our complete benchmark. These models have different execution profiles which impacts significantly on the overall performance. The main difference is the time expended on the backward traversal phase, i.e. the cycle detection phase.

Figure 4 gives a series of bar charts where we put in evidence the time required for each phase of the algorithm (exploration and cycle detection). In addition, we compare our approach (RG and RPG) with a "standard" algorithm for model-checking CTL using 16 processors on our test machine. For this standard algorithm, we can simply use the same code than for the RG version, but compute the predecessor relation instead of relying on the reverse graph. Since we do not need to store the transition relation, we call this new version of our algorithm NO_GRAPH.

We have observed two main categories of behaviors in this analysis. We have examples of *complete backward traversal* and examples of *negligible backward traversal*.

Negligible backward traversal We put in this category the examples where the time spent in the backward exploration phase is negligible compared to the overall execution time (see Token Ring model in Figures 3 and 4). This is the case, for instance, if the specification is false and the cycle detection phase terminates early. In this category of experiments, there is no significant differences between RG and RPG. This is mainly because the gain in performance during the forward exploration phase outweighs the extra work performed during the cycle detection phase.

Complete backward traversal. We put in this category the examples where the cycle detection phase needs to run through all the state space ((see Solitaire model in Figures 3 and 4)). We observed a significant difference in performance between the RG and RPG versions in this case. The extra work performed by the RPG version becomes the dominant factor, up to a point where it accounts for nearly all the execution time.

A real advantage of the RPG version is to impose no restrictions on the models that are checked. The NO_GRAPH algorithm rely on the fact that the transition relation needs not be stored. Very often, this optimization is based on the fact that it is possible to compute the reverse transition relation. But this is not always practical, or even possible.

To conclude, both RG and RPG can be useful; RPG being the good choice if we are limited by the memory space. Although RPG may requires a lot more computations, it can be applied on models that are not tractable with the reverse graph version. For instance, we performed an experiment with the European Peg-Solitaire game (37 pegs) with our setup (208 GB of RAM). The state space of this model has 3.10^9 states and 3.10^{10} transitions. Assuming that each transition would use 8 bytes of memory to store the reverse relation between two given vertices, we would need at least 240 GB of memory only to store the edges of this graph. On the opposite, we only need 15 GB to store the states and we can check this example with RPG (with the specification `A<> dead`) in 19,662 s, divided in 3,817 s for the exploration phase (less than 20% of the computation time) and 15,845 s for the cycle detection phase.

6 Comparison With Other Tools

We present a comparison of our algorithms with DIVINE [1], which is the state of the art tool for parallel model checking. More recently, Barnat et al. published

that the conjugated use of owcty and map (see section 2) results in a optimal on-the-fly algorithm for the verification of weak LTL properties. The result of this union (map-owcty) is an algorithm that first tries to disproof a formula using the map strategy until it ends the first iteration, otherwise it proceeds with the owcty algorithm. Unfortunately, this algorithm is not yet available for use on the latest distribution. (The results reported here were obtained using the DIVINE 2.5.2 version.) Consequently, for this benchmark, we consider owcty and map separately. It does not affect our analysis because the union map-owcty tries to bring together the best from both in one algorithm. Thus, we could consider that DIVINE holds a better performance whenever one of these two algorithms has the best result.

This benchmark is based on the set of models presented in [1], we selected the models which there were available valid and non valid formulas. This choice was motivated to establish a broader comparison between our approach, which is "optimized" for valid formulas, and theirs, which are on-the-fly algorithms. Like we mentioned, RG and RPG are not completely on-the-fly algorithms because a cycle is detected after the state spaced is constructed, what can delay the discovery of an invalid path. By contrast, owcty and map are meant to generate the complete state space when they are not able to disproof the formula, i.e., the formula is valid. That means that they are more likely to find invalid paths faster than our approach.

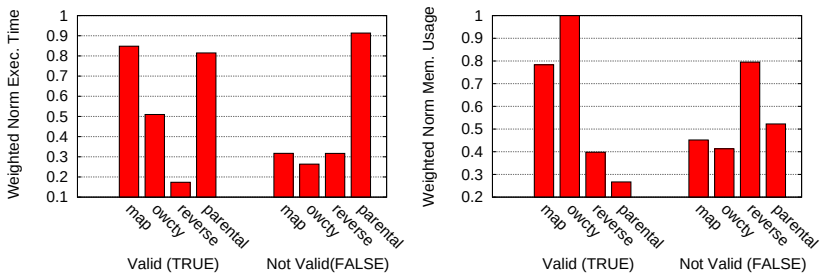


Fig. 5. Comparison with divine.

Figure 5 summarizes the results obtained in this benchmark; it shows a set of histograms for the normalized weighted sum of the memory footprint and the execution time. These measures are first normalized and after weighted by the number of states of each experiment. We presented this measure in order to balance the results according to the size of the model. The results are divided by the type of the formula (Valid or not valid) and the tool (or algorithm) used. From this figure, owcty/maps stands for the results obtained using the DIVINE tool and reverse/parental for RG and RPG algorithms implemented on Mercury. The complete benchmark is presented in the Appendix A.

As expected, owcty and map have a better overall outcome whenever the formula is not valid (FALSE). By contrast, reverse holds the best execution time when the formula is valid. This is due to the linear complexity of RG when compared to the other solutions. Regarding parental, our results show

that it holds the best memory footprint among all results, in average, it uses 2—4 times less memory than map and owcty when the formula is valid. In addition, regardless of its non-linear time complexity, it showed good results when compared to map and owcty. For instance, it is able to verify a valid formula —in average— using 4 times less memory than owcty by a small amount of extra time ($\approx 60\%$ more).

To conclude, for the set of models and formulas used in this benchmark, both parental and reverse delivered good results when compared to DIVINE. For instance, reverse presented a better performance in both time and memory usage when compared with DIVINE (map and owtcy). Moreover, parental proved its *economical memory profile* by using less memory than reverse and DIVINE.

7 Conclusions

In this work, we have described some ongoing work concerning parallel model-checking algorithms for finite state systems.

We define two versions of a new algorithm, called MCLCD, that supports specification expressed in a subset of CTL. Our algorithms are based on a standard, semantic model-checking algorithm for CTL that specifically targets parallel, shared memory machines. We defined two versions of the same algorithm: a Reverse Graph (RG) version, that explicitly stores the transition relation in memory; and a Reverse Parental Graph (RPG) version, that only requires a "spanning subtree" of the transition relation.

We use the reverse parental graph structure as a mean to fight the state explosion problem. In this respect, this approach has a similar impact in space than algorithmic techniques like *sleep sets* (used with partial orders methods), but with the difference that we do not take into account the structure of the model. Moreover, our approach is effective regardless of the formalism used to model the system.

Our prototype implementation shows promising results for both the RG and RPG versions of the algorithm. The choice of a "labeling algorithm" based on the out-degree number has proved to be a good match for shared memory machines and a work stealing strategy; for instance, we consistently obtained speedups close to linear with an average efficiency of 75%. Our experimental results also showed that the RPG version is able to outperform the RG version for some categories of models.

For future works, we are studying an improved version of our algorithms that supports the complete set of CTL formulas without manage several copies of our labels (sons and suc) in parallel; it could have an adverse effect on the memory consumption.

References

1. J. Barnat, L. Brim, and P. Rockai. A Time-Optimal On-the-Fly parallel algorithm for model checking of weak LTL properties. In *Formal Methods and Software Engineering (ICFEM 2009)*, volume 5885 of *LNCS*, page 407425. Springer, 2009.

2. Alexander Bell and Boudewijn R. Haverkort. Sequential and distributed model checking of petri nets. *International Journal on Software Tools for Technology Transfer*, 7(1):43–60, April 2004.
3. Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the Alternation-Free -Calculus. In Dragan Bosnacki and Stefan Leue, editors, *Model Checking Software*, volume 2318 of *Lecture Notes in Computer Science*, pages 501–522. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46017-9_11.
4. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
5. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, volume 131, pages 52–71. Springer-Verlag, Berlin/Heidelberg, 1982.
6. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2):275–288, 1992. 10.1007/BF00121128.
7. Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, USA, 1995.
8. G. J Holzmann, R. Joshi, and A. Groce. Swarm verification. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, page 16, Washington, DC, USA, 2008. IEEE Computer Society.
9. Cornelia P. Inggs and Howard Barringer. CTL* model checking on a shared-memory architecture. *Formal Methods in System Design*, 29(2):135–155, July 2006.
10. A. W. Laarman, R. Langerak, J. C. van de Pol, M. Weber, and A. Wijs. Multi-Core nested Depth-First search. In T. Bultan and P-A. Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis, ATVA 2011, Tapei, Taiwan*, volume online pre-publication of *Lecture Notes in Computer Science*, London, July 2011. Springer Verlag.
11. P. Merlin and D. Farber. Recoverability of communication ProtocolsImplications of a theoretical study. *Communications, IEEE Transactions on*, 24(9):1036 – 1043, September 1976.
12. Jaco van de Pol and Michael Weber. A Multi-Core solver for parity games. *Electronic Notes in Theoretical Computer Science*, 220(2):19 – 34, 2008. Proceedings of the 7th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2008).
13. John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.
14. Rodrigo Saad. *Parallel Model Checking for Multiprocessor Architecture*. PhD thesis, L'Institut National des Sciences Appliquées de Toulouse, Toulouse - France, 11/2011 2011.
15. Rodrigo T. Saad, Silvano Dal Zilio, and Bernard Berthomieu. Mixed Shared-Distributed hash tables approaches for parallel state space construction. In *International Symposium on Parallel and Distributed Computing (ISPDC 2011)*, page 8p., Cluj-Napoca, Romania, July 2011. Rapport LAAS ngre 11460.
16. Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114 –121, October 1971.

A Results

In this Appendix, we give the set of models, with their individual results, used for the comparison between our algorithms and Divine. Concerning this benchmark, all the configurations (owcty, map, reverse/RG and parental/RPG) were performed using all the available resources of our setup. All the experiments were carried out using 16 cores and with an initial hash table size enough to store all states. The DIVINE experiments were executed with an addition flag (*-n*) to remove the counter-example generation for performance purpose.

Figure 6 depicts the set of models used for this comparison.

Model	Formula	Results
Anderson (AN) $18 \cdot 10^6$ states	F1:`(-cs_0) ==>  (cs_0)`	*false*
	F2:`A[]<>(cs_0 or ... or cs_n)`	*true*
Lamport (LA) $38 \cdot 10^6$ states.	F1:`(wait_0 and (- cs_0)) ==> (cs_0)`	*false*
	F2:`(- cs_0) ==> (cs_0)`	*false*
	F3:`A[]<>(cs_0 or ... or cs_n)`	*true*
Rether (RE) $4 \cdot 10^6$ states	F1:`A[]<>(nrt_0)`	*true*
	F2:`A[]<>(rt_0)`	*false*
Szymanski (SZY) $2 \cdot 10^6$ states .	F1:`(wait_0 and (- cs_0)) ==> (cs_0)`	*false*
	F2:`(- cs_0) ==> (cs_0)`	*false*
	F3:`A[]<>(cs_0 or ... or cs_0)`	*true*

Fig. 6. Formulas and Models for our Comparison.

Figure 7 presents the results obtained for each model presented at Figure 6. This table starts by presenting the results for the anderson model, followed by the lamport model, the rether model and finally the szymanski model. For each model we give the execution time (T.) in seconds and the memory peak (M.) (in Gb).

M	Formula	owcty		map		reverse		parental	
		T.(s)	M.(Gb)	T.(s)	M.(Gb)	T.(s)	M.(Gb)	T.(s)	M.(Gb)
AN	F1: *false*	61.3	3.3	110.2	5.5	28.8	2.8	94.4	1.8
	F2: *true*	79.5	7.4	110.5	4.8	26.4	2.9	50.4	1.8
LA	F1: *false*	1.6	1.1	1.4	1.1	42.4	5.1	74.2	3.3
	F2: *false*	1.4	1.1	1.7	1.2	47.6	5.6	327.2	3.6
	F3: *true*	153.6	14.1	282.8	12.1	51.0	5.6	370.4	3.7
RE	F1: *true*	12.0	1.8	20.1	1.3	5.0	0.7	12.0	0.6
	F2: *false*	13.2	1.8	1.2	0.3	3.4	0.7	7.8	0.6
SZY	F1: *false*	8.5	0.9	7.0	0.5	2.2	0.3	1.4	0.2
	F2: *false*	9.8	0.9	6.6	.5	4.2	0.3	39.6	0.3
	F3: *true*	9.0	0.9	24.7	0.6	3.8	0.3	32.8	0.3

Fig. 7. Table of results.