Parallel Model Checking With Lazy Cycle Detection MCLCD*

Rodrigo T. Saad, Silvano Dal Zilio and Bernard Berthomieu

CNRS, LAAS, 7 avenue du Colonel Roche, F-31400 Toulouse France
Univ de Toulouse, LAAS, F-31400 Toulouse, France
{rsaad, dalzilio, bernard}@laas.fr

In this work, we present new algorithms for exhaustive parallel model checking that are as efficient as possible, but also "friendly" with respect to the work-sharing policies that are used for the state space generation (e.g. a work-stealing strategy): at no point do we impose a restriction on the way work is shared among the processors. This includes both the construction of the state space as the detection of cycles in parallel, which is is one of the key points of performance for the evaluation of more complex formulas.

1 Introduction

Model Checking is a valuable formal verification method that can be used to avoid the presence of logical errors. Roughly speaking, it has the same impact as performing an exhaustive test, using symbolic evaluations, where every possible scenario is checked for correctness.

Model checking has emmerged 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. So large indeed that is goes beyond the available computing resources. The state explosion problem is one of the main challenges faced by model checking researchers.

Despite the fact that considerable progress have been made—such as symbolic model checking or partial-order techniques—there are still classes of systems that cannot benefit from these progress on the algorithmical side. For these systems, a classical—exhaustive state—enumerative approach is still the most appropriate.

In this work, the idea is to take benefit of recent advances on the hardware side to improve enumerative model checking techniques. Indeed, we now have access to computers with larger shared memory space and to multi-core architectures that makes feasible the verification of larger models, in a reasonable amount of time.

In this context, 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 [TSDZB11]. By compatible, we mean that we base our approach on the same set of hypotheses; actually, we should say the same absence of restrictions. First, we follow an enumerative, explicit-state

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

approach. We assume that we are in the least favorable case, where *we have no restrictions on the models that can be analyzed*. For instance, we cannot rely on the existence of a symbolic representation for the transition relation, such as with symbolic model-checking. Next, *we make no assumptions on the way states are distributed* over the different local dictionaries (we assume the case of a non-uniform, shared-memory architecture). Finally, *we put no restrictions on the way work is shared among processors*, that is to say, the algorithm should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing (see the discussion in Sect. 3).

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 LTL[1]. These algorithms rely on efficient methods to detect the presence of cycles in a graph, such as Tarjan's algorithm [Tar71] or "nested-DFS" [CVWY92]. 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. Indeed, it is known since the 1980's that "depth-first search is inherently sequential" [Rei85], more precisely, that it is related to problems that are P-space complete. 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.

We can give a second justification, that is more related to the implementation choices made in our work. Indeed, algorithms that rely on exploring the state space graph in a specific order go against our requirement that the state space exploration should be friendly to traditional work-sharing techniques.

Based on these observations, we decided to follow an alternative approach that we call *semantic model-checking*. This is the approach initially proposed by Clarke and Emerson [CE82, Cla99] for 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. Similar algorithms are also used for evaluating modal μ-calculus formulas [EJS93] (if we leave out the extra-complexity involved with modal fixpoint alternation).

To obtain an efficient parallel model-checking algorithm, we decided to limit ourselves to a restricted subset of temporal logic formulas. We made the choice of a subset of formulas, expressible both in CTL and LTL, that is equivalent to the requirement specification language supported by Uppaal [BDL04]. This subset includes formulas for expressing basic invariant and reachability properties, but also more complex properties, such as $\psi \rightsquigarrow \phi$, meaning that every state where ψ holds eventually "leads to" a state where ϕ holds.

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 can avoid the case of dead states—states without successors—by adding a trivial loop to each such state.) It should be observed that, since we work with finite state systems, any infinite path includes at least one cycle. This remark explains why, in

[1] We use the term automata-theoretic to denote algorithms in which model-checking is reduced to composing the system with an automaton that accepts traces violating the specification; and then using graph algorithms to search for a counterexample trace.

the remainder of this chapter, we will often focus our attention on the problem of identifying a cycle in a graph.

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[2] 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 for parallel Model Checking. Next, Section 3 introduces the contributions of this work. In Section 4, we define the set of logical formulas supported by our approach. After the definition of basic results on graph theory in Section 5, we describe our parallel algorithms using pseudo-code in Section 6. We study the fundamental properties—complexity, termination, soundness, ...—in Section 7. Before concluding, we give a set of experimental results performed with our prototype implementations in Section 9 and a comparison with the state-of-the-art tool DiVine in Section 10.

2 Related Work: Parallel Model Checking

In this section, we present all the related work for parallel Model Checking. Even though we are interested for shared memory architecture, we also decided to present the literature for distributed memory machines because they can be easily implemented into shared memory machines, indeed some of the available solutions we have nowadays were first developed targeting distributed memory machines. We present the solutions for parallel model checking according to the kind of properties they support, Section 2.1 presents the related solutions for LTL and Section 2.2 for CTL parallel Model Checking.

2.1 Parallel LTL Model Checking

In this section we present the related literature for parallel LTL model checking. The parallel algorithms that supports Linear Temporal Logic formulas follow the sequential Automata-Theoretic approach [Wol86]. The difficulty in this case is to determine whether an accepting state is part of a cycle.

One of the most efficient solution for sequential Model Checking uses the Tarjans algorithm [Tar71] to find all strongly connected components (SCC); the emptiness problem is reduced to check if an accepting state is part of a SCC; it can be done in linear time in the size of the system but it may incur in extra memory consumption since the states must be stored explicitly for the computation of SCCs.

Another efficient solution and broadly used is [CVWY92] which uses a nested depth first search (Nested-DFS) algorithm to find if an acceptance state is reachable from himself; every accepting state found by the first search triggers a second one to find if it is part of a cycle.

Although both Tarjans and Nested-DFS algorithms are efficient and simple to implement, they are hard to perform in parallel because they depend on the sequential depth-first search post-order of vertices.

[2] In the context of this work, we prefer to use the inverse of the transition relation because we propagate labels from a state to its parents.

It can be illustrated with the example taken from [BBS01] and presented in the Figure 1. It is not possible to guarantee that the cycle through the accepting node *C* will be found by the Nested-DFS algorithm if the parallel DFS exploration does not preserve the DFS order of visited states. For instance, if the nested search starts first from node *A*, node *C* will be flagged and the cycle ignored. This problem is inherently sequential and cannot be performed efficiently in parallel (see [Rei85] for a complete discussion).

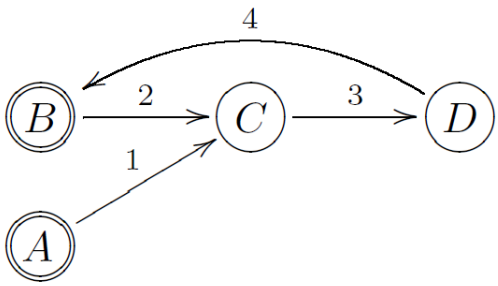


Figure 1: Example of the sequential depth-first search post-order of vertices. [BBS01]

The difficulty to find good parallel algorithms for LTL Model Checking can be linked to the fact that "finding a cycle in a graph is an inherently sequential problem" (a claim linked to the fact that the *edge maximal acyclic subgraph* problem is P-complete, see problem A.2.18 in [GHR95]). In this section we present all relevant solutions for parallel LTL Model Checking according to theirs characteristics, we divided them into DFS order and BFS/arbitrary order solutions.

None of the parallel solutions proposed for LTL Model Checking are so efficient like the sequential ones because of the difficulty to assert that an accepting state is part of a cycle. In this section we present all relevant solutions for parallel LTL Model Checking according to theirs characteristics, we divided them into DFS order and bfs/arbitrary order solutions.

DFS order

The first parallel algorithm for LTL Model Checking was presented by Barnat et al. [BBS01] and was based on a previous work from Lerda et al. [LS99] to deploy the Spin Model Checker on a cluster. Barnat et al. instrumented their exploration engine with a *dependency structure* in order to keep the sequential depth first post-order for the evaluation of accepting cycles: "A nested DFS procedure is allowed to begin from a seed S if and only if all seeds below S have already been tested for cycle detection" [BBS01]. This strategy to preserve the sequential post-order comes with the price of undesired synchronizations for the *dependency structure*, moreover, only one nested DFS procedure is allowed at a time. This work targets distributed memory machines (NOW cluster) and makes use of a "generic" static function to distribute work; it was designed to tackle models otherwise untreatable for a single workstation. Hence, execution time (speed up) is not the main concern and only results about the capacity to check bigger models are reported.

The inherent difficulties to keep the sequential DFS order had driven some authors to try a different approach in order to keep the nested search "local", i.e., in a single node (or processor). In [BBC02] and [Laf02], a special static partition function based on the never claim automaton (negation of the Buchi Automaton) is used to "localize" cycles. The main objective is to "... distribute states such that all states of a strongly connect component (or equivalently a cycle) belong to the same equivalence class ..." [Laf02]. Hence, all states that belongs to a given cycle will be part of the same *equivalence class*.

Assigning each of these classes to only one node is enough to ensure correctness for the nested DFS since the sequential DFS traversal order is still respected within an *equivalence class*. The main drawback of this technique is that LTL formulas do not hold a significant number of SCC and, by consequence, do not generate a sufficient number of *equivalence class*. (For instance, some formulas may hold only one SCC.)

Next we give some information about three important algorithms that belong to the category of DFS algorithms.

[spin multi-core] Another relevant work that seeks for a simpler approach is [HB07] which uses the notion of *irreversible transitions* to partition the state space into disjoint subsets. This work was specifically conceived for the multi-core technology and proposes a dual-core extension of the Spin model checker. They use these so called *irreversible transitions* to trigger the nested DFS search; these transitions give two approximately equal and disjoint sets of states when they separate the first from the second search. This design uses only two CPUs, one for each DFS search, and use the *irreversible transitions* to send work from the first to the second CPU. The main advantage of this work is that "... The complexity remains linear in the size of the number of reachable system states, with the same constant factor as applies to the standard nested depth-first search". Although this approach is not scalable for more than two processors, the authors present it as simple and effective solution for dual-core (two CPUs) machines.

[swarm] Another solution for distributed LTL model checking was proposed by Holzmann et al. in [HJG08b, HJG08a]. The idea is to have multiple independent instances (workers) of the problem following different exploration heuristics. The main objective is to find errors quickly and not to verify the complete state space. The design is simple, each worker follows a different exploration strategy—i.e., a DFS, BFS or a random order of exploration—until one of them finds an error.

[mc-ndfs] Recently, **swarm** had been extended for multi-core architectures in [LLP+11] and named *multi-core nested DFS* (**mc-ndfs**). They propose 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. The basic idea is to share information in the backtrap of the nested search. Thus, when a worker backtracks from the nested search, the state involved will be globally ignored through a global coloring scheme, pruning the search spaces for all workers i. Even if in the worst-case each processor might still traverse the whole graph ($O(N \cdot (S+R))$ where N is the number of processors), 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.

Concerning the reported results, as expected, **mc-ndfs** is fast and scales well whenever there is an accepting cycle, otherwise, it suffers a significant loss of performance. The authors justify this loss of performance due to the lack of coloring sharing, "... in the worst case (no accepting cycles) no work is shared between " processors.

BFS/Arbitrary order

The research group that develops DiVinE is one of the most active team and one of the few to propose a different approach for parallel LTL model checking. They introduced a series of four algorithms [BCKP01, BBC03, BCMS04, CP03] not based on the DFS order, these algorithms are indeed based on breath first or arbitrary exploration order. Although they were initially developed for distributed memory machines, the DiVinE team, in [BBR07], reused these algorithms in a search for a good candidate for shared memory machines.

[negc] The first algorithm to follow a different exploration order was presented in [BCKP01] based

on the *negative cycles detection* (**negc**). The problem is reduced to finding negative length cycles in the directed graph obtained from the synchronization of the system with a weighted Buchi automaton. In this case, all edges out-coming from accepting states are assigned with -1 and all others with 0, consequently, "negative cycles will simply coincide with accepting cycles and the problem of Buchi automaton emptiness reduces to the negative cycle problem" [BCKP01]. They propose a distributed method to solve this problem and compare the results with their previous distributed nested DFS algorithm [BBS01]. Despite the fact that it has a theoretical worst-case time complexity in $O(R \cdot S)$, the algorithm outperformed [BBS01] (DFS based) because this new algorithm allows for a higher degree of asynchronous parallelism, instead of all the strict synchronizations imposed by the *dependency structure* in [BBS01].

[bledge] Later, Barnat et al., in [BBC03], proposed a distributed memory algorithm named *back-level edge* (**bledge**) based on computing the distance between a vertex, u, and the root following a BFS exploration. This distance is denoted $d(u)$. Indeed, a necessary condition for a path in a graph to be a cycle is that it has two vertices, u and v, following each other such that v is closer to the source than u; an edge (u,v) is called a back-level edge if and only if $d(u) \geq d(v)$. The algorithm computes the BFS distance to detect back-level edges and then check the presence of cycles following a DFS search. The algorithm has a time complexity of $O(S \cdot (S+R))$ and requires some synchronizations to ensure that incorrect distances are never assigned to vertices due to the non-deterministic character of the exploration order in parallel.

[map] Another work that follows the BFS exploration order is the *maximal accepting predecessors* (**map**) [BCMS04]. This work is based on the observation that all vertices on a cycle have exactly the same predecessors: if two vertices belong to the same cycle then they have the same set of predecessors and they belong to this set.

It is not necessary to consider the complete set of vertices in this case, since we are only interested by the accepting states of the automaton. Moreover, it is not necessary to consider all the predecessors of a vertex; the algorithm proposes to use a representative subset of states, called the *maximal accepting predecessors*, obtained from a presupposed linear ordering of vertices.

A drawback is that the set of maximal accepting predecessors may have to be updated dynamically. An advantage is that the algorithm is not bound to a specific traversal order, it does not depend on the sequential DFS postorder. The algorithm work in iterations and has a time complexity of $O = (A^2 \cdot S)$, where A is the number of accepting states.

[owcty] Cern et al. in [CP03] proposed an algorithm to detect fair cycles that has linear time for weak LTL specifications but has a quadratic worst case complexity. In brief, "the language of the automaton is nonempty if and only if the graphs corresponding to the automaton contains a reachable *fair cycle*, that is a cycle containing at least one state from every accepting set ...". It was inspired from the symbolic algorithm called *One Way Catch Them Young* (**owcty**) [FFK$^+$01], it first computes an approximation of the set of states that contain all fair components and after refines it by successively removing unfair components until it reaches a fixpoint. Although it has a better time complexity in average when compared to the algorithms based on the nested DFS, this work is not on-the-fly and requires several synchronizations in order to refine the set of states. It has a time complexity of $O = (h \cdot (S+R))$ where h is the height of the SCC quotient graph.

The results reported using the distributed version of the **owcty** algorithm could not match the performance of the distributed nested DFS [BBS01] mainly because of the amount of messages and synchronizations required by the distributed implementation. However, it is a good choice for shared memory machines because there is no need to exchange messages and the synchronizations can be achieved just by using barriers and lock regions; a prototype version is presented in [BBR07] and the results reported (see figure 2) are very good when compared to other similar algorithms [BCMS04, BBC03, BCKP01].

Moreover, [BBR07] addresses the analysis of a shared memory candidate for LTL model checking; all these algorithms were originally developed for distributed machines.

[owcty + map] The state of the art algorithm for parallel LTL model checking was presented by Barnat et al. in [BBR09], they proposed an on-the-fly algorithm 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.) Since **owcty** requires the complete exploration of the state space to work, the accepting cycle detection procedure from **map** is employed but without the re-propagation of accepting predecessors in order to maintain the time complexity linear. On the other hand, if the limited **map** procedure fails to find a accepting cycle, the rest of the algorithm executes the original **owcty**.

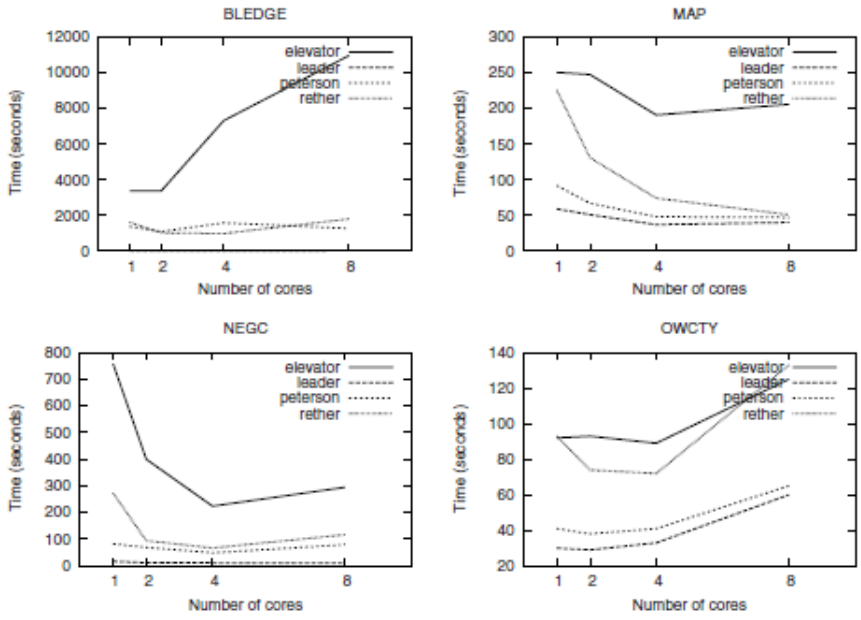


Figure 2: Parallel LTL algorithm comparison for shared memory machines. [BBR07]

Until recently, the results obtained with **owcty + map** indicated that the most appropriate parallel on-the-fly solution would be an heuristic algorithm following a different exploration order than DFS. However, new algorithms such as **mc-ndfs** are showing that under some circumstances, it is possible to achieve a better result following an adapted nested DFS approach. [LLP+11] depicts a comparison between **mc-ndfs** and **owcty + map** for models with and without cycles. For models with accepting cycles, **mc-ndfs** finds counter-examples faster than **map+owcty** due to its depth-first on-the-fly nature, otherwise, in some cases **owcty + map** is faster with a factor of 10 on 16 cores. It is still early to state which one is the best. Meantime, we would like to remind that the worst case scenario for **mc-ndfs** is in fact the case when the formula is valid, hence, this solution is more adapted to find errors fast and not to check if the property is valid.

2.2 Parallel CTL Model Checking

In this section, we present the parallel explicit algorithms for CTL model checking. In addition, we also present parallel algorithms for model-checking the alternation free μ-calculus, denoted L^1_μ, because of its close relationship to CTL.

CTL model checking can be classified into global and local algorithms. Global algorithms are more suitable to be executed in parallel because cycles can be detected later, indeed, they do not depend on the sequential DFS exploration order because cycles are checked after the complete construction of the state space. While local algorithms decides the problem through a depth-first search, they have the advantage of being on-the-fly.

The first work proposed in this context was for L^1_μ, presented by Bolling et al. in [BLW01], in terms of games [EJS93, Sti99] where the moves correspond to paths in the "game graph" (the Cartesian product of the states s with the formula ϕ). Similar approaches are presented in [BCY02] and, reformulated in terms of solving alternating boolean equation systems, in [JM05].

Bell et al., in [BH04], propose a distributed version of [CES86]. It is based on a labeling procedure and uses the parse tree of the CTL formula to evaluate sub-formulas. The labeling procedure is carried in a distributed manner, a given state is labelled only by its owner following the static slicing strategy used to distribute the states.

In [BLW02], Bolling et al. extend their previous work [BLW01] and propose a parallel local algorithm for L^1_μ which circumvents the sequential DFS limitation of their initial algorithm by omitting the detection of cycles. This work follows the same idea of "coloring a game graph" but defines a backward color propagation process. All these works were targeting distributed memory machines and use a static partition to construct the game graph in parallel.

In contrast with the number of solutions proposed for distributed memory machines, just two solutions were conceived for shared memory machines.

[pmc] Inggs et al. [IB06] give the first work specifically developed for shared memory machines and implemented as part of the automata-driven parallel model checker called PMC. It is similar to [BLW01] but it differs in the way winning positions are determined. They use a formalism called Hesitant Alternating Automata [Kup95] to represent the formula specified in CTL*. New games are played locally every time a game position is revisited in order to decide if it is an infinite play (cycle) or part of a different path. For implementing this algorithm, the authors decided not to use locks and therefore to accept a duplication of work.

[[PW08]] Finally, there is [PW08] for shared memory machines based on game graphs. It proposes a parallel algorithm to solve the two-player parity games [Wil01], which is equivalent to model checking for μ-calculus.

3 Contributions

We presented in Section 2.1 and 2.2 an overview of the literature for parallel model checking. We tried to put in evidence the need for an effective parallel cycle detection strategy in order to obtain an efficient parallel algorithm. The verification of more elaborated formulas requires the use of efficient algorithms to detect the presence of cycles, such as the use of the Strong Connected Components (SCC for short) abstraction, i.e. [Tar71], or the "nested-DFS" [CVWY92] algorithm. These algorithms are hard to parallelize because they heavily rely on following a particular order when exploring the state graph (for example a DFS order).

We believe that searching only for efficient[3], on-the-fly, parallel solution miss part of the problem. Indeed, to obtain a good complexity in practice, it is also important to able to benefit from useful work-sharing policies during the complete model-checking process; not only during the state space construction.

With the same idea to focus on "pragmatic" optimizations, we also focus on approaches that requires less memory space. Our main goal is to propose a solution that is suitable for any state class abstraction without taking into account neither the structure of the model nor its symmetry.

We make the following contributions in the domain of algorithms for parallel model checking.

We define a new algorithm, that we call MCLCD for Model Checking Algorithm with Lazy Cycle Detection. This algorithm is "compatible" with the parallel state space generation techniques we described in [TSDZB11]. By compatible, we mean that we base our approach on the same set of hypotheses; actually, we should say the same absence of restrictions. First, we follow an enumerative, explicit-state approach. We assume that we are in the least favorable case, where *we have no restrictions on the models that can be analyzed*. For instance, we cannot rely on the existence of a symbolic representation for the transition relation, such as with symbolic model-checking. Next, *we make no assumptions on the way states are distributed* over the different local dictionaries (we assume the case of a non-uniform, shared-memory architecture). Finally, *we put no restrictions on the way work is shared among processors*, that is to say, the algorithm should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing.

We propose two versions of MCLCD: a first version based on a reverse traversal of the state graph, called RG, where we need to explicitly store the transtions 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 $O(|S|)$.

The main advantage of the RPG version is to provide an algorithm that is efficient in memory and independent of the choice of states classes abstraction. Figure 3 lists our algorithms among related solutions for parallel model checking that have already been implemented on shared memory machines.

Algorithm	Time Complexity	Logic Supported
map	$O(\|A\|^2\cdot\|S\|)$	LTL
owcty	$O(h\cdot\|S\|)$	LTL
negc	$O(\|R\|\cdot\|S\|)$	LTL
bledge	$O(\|S\|\cdot(\|S\|+\|R\|))$	LTL
mc-ndfs	$O(N\cdot(\|S\|+\|R\|))$	LTL
MCLDCD-RG	$O(\|S\|+\|R\|)$	sub-CTL
MCLCD-RPG	$O(\|S\|\cdot(\|R\|-\|S\|))$	sub-CTL

Figure 3: Parallel model checking algorithms for shared memory machines.

Our algorithms follow the classical semantic approach proposed by Clark et al. in [CES86], with the distinction that we only support a subset of CTL formulas. (We say in Sec. 11 how this restriction could be lifted). We follow an approach based on labeling states, like with [BH04] and [BLW01, BLW02] in the context of game automaton. We chose a semantic approach because we believe that it is more appropriate for a parallel algorithm with dynamic work-load strategies.

[3]Efficient in the sense of "with a good theoretical, worst-case complexity"; such as linear in the size of the state graph for example.

Finally, we implemented our algorithms using the work-stealing strategy [TSDZB11] both for the state space construction phase and the property validation (cycle detection) phase. Related works [HB07, LvdPW10, IB02] use dynamic workload policies for the parallel state space construction only, they do not employ any kind of work-load approach during cycle detection: Holzmann et al. perform the nested search in a exclusive allocated process; 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.

4 List of Supported Properties — the LRL Logic

Model-checking is a problem with two variables: it depends both on (1) the formal language used to express the models, and (2) the theoretical framework used to express the specification of the system. In this work, we follow a classical approach in which properties are expressed using temporal logic formulas.

Several approaches for model checking exists in the literature, supporting a different set of temporal logics. Linear Temporal Logic (LTL) and Computation tree logic (CTL) are the most popular options, mainly because there exist efficient algorithms; algorithms with a linear time complexity in the size of the state graph.

While sequential algorithms for model-checking are a well-studied research subject, we believe that it is still early to claim that the problem is settled in the parallel case. Indeed, to the best of our knowledge, there are no good parallel algorithm that: covers a "complete logic" (such as the whole of LTL or CTL); has a linear, worst-case complexity which does not duplicate work; and provides a good speedup independent if the property holds or not.

In the context of our work, we do not try to define a parallel algorithm with all these good properties and favor speedup. We decided to trade off the expressiveness of the specification language for a better parallel complexity. Furthermore, in the case of the second version of our algorithm, we are willing to abandon our requirement over the linear, worst-case complexity of the algorithm. We show with our experimental results that this may prove a good choice in practice.

We limit ourselves to a restricted subset of temporal logic formulas that corresponds to the requirement specification language supported by Uppaal [BDL04]. This 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; that is to say, we are interested by properties that are valid for the initial state, and not from any other given state.

We give the list of supported properties below, with a simple explanation for each property. We use the symbols $\phi, \psi, \ldots$ to denote *predicates*, that is statements that may be true or false depending only on the state on which they are evaluated[4].

- $E\Diamond(\phi)$: this property express the *possibility* for ϕ to hold. The property is valid if there is a path, starting from the initial state, that reach a state where ϕ holds. This corresponds to the formula $EF\,\phi$ in CTL (note that we are only interested by the validity of this formula for the initial state).

[4]Since we define a general model-checking algorithm, we do not specify exactly what is the language of predicates. This parameter may be changed depending on the underlying languages used to define the models.

- $E\Box(\phi)$: this property express that ϕ holds *potentially always*. The property is valid if there is an infinite path, starting from the initial state, where ϕ holds for every state s in the path. This corresponds to the formula $EG\,\phi$ in CTL.
- $A\Diamond(\phi)$: this property express that ϕ always *eventually* holds. The property is valid if, for every path starting from the initial state, there is a state where ϕ holds. This corresponds to the formula $AF\,\phi$ in CTL. The property $A\Diamond(\phi)$ is true if $E\Box(\neg\phi)$ is false.
- $A\Box(\phi)$: this property express that ϕ is an *invariant*; it always holds. The property is valid if, for every reachable state, ϕ holds. This corresponds to the formula $AG\,\phi$ in CTL. The property $A\Box(\phi)$ is true if $E\Diamond(\neg\phi)$ is false.
- $\psi \rightsquigarrow \phi$: the *leadsto* property express that from every state where ψ holds, eventually ϕ will hold. This corresponds to the formula $AG\,(\psi \Rightarrow AF\,\phi)$ in CTL. This is the only property that corresponds to a CTL formula with nested modalities.
- $E\,(\psi \cup \phi)$: the *reachability* property is valid if there is a path, starting from the initial state, such that ψ holds until we reach a state where ϕ holds. This is an extension of the possibility property since $E\Diamond(\phi)$ is equivalent to $E\,(\text{True} \cup \phi)$. This property is not part of Uppaal's specification language and encompasses the formulas $E\Diamond(\phi)$ and $A\Box(\phi)$.
- $A\,(\psi \cup \phi)$: the *liveness* property is valid if for every path, starting from the initial state, the predicate ψ holds until we reach a state where ϕ holds. This is an extension of the eventuality property since $A\Diamond(\phi)$ is equivalent to $A\,(\text{True} \cup \phi)$. This property is not part of Uppaal's specification language and encompasses the formulas $A\Diamond(\phi)$ and $E\Box(\phi)$.

We call our specification language LRL, for Leadsto-Reachability-Liveness. As we said in the description of LRL, every property corresponds to an equivalent CTL formula. LRL is strictly less expressive than CTL. In particular, the language does not allow the nesting of formulas (modalities are always applied to predicates). Nonetheless, we believe that our specification language is expressive enough in many practical cases; the choice of this language by the designers of Uppaal gives at least some support to the claim that this subset provides a good balance between expressiveness and efficiency.

Formal Semantics of LRL. Next, we define the semantic of our specification language with respect to a given *Kripke structure*, KS, using an entailment relation. The definition is very similar to the traditional presentation of the semantic of CTL formulas. We use a Kripke structure $KS = (S, R, s_0)$ to represent the behavior of the system. We use S to denote the set of states, $s_0 \in S$ to denote the initial state, and R to denote the transition relation between states in E. We use the symbol τ to define an infinite sequence of states (a trace) $s_0 \cdot s_1 \cdot \ldots$ such that $s_i R s_{i+1}$ for all i. Finally, we use the notation $(S, R, s_0) \vDash F$ to denote that formula F holds for the Kripke structure (S, R, s_0) and $s \vDash \phi$ to denote that the predicate ϕ holds for s. Considering the different equivalences between properties that we already gave, we simply need to define the semantics for the formulas $E\,(\psi \cup \phi)$, $A\,(\psi \cup \phi)$, and $\psi \rightsquigarrow \phi$.

$$
\begin{array}{rll}
(S,R,s_0) \vDash E(\psi \cup \phi) & \text{iff} & \text{for some path } \tau = s_0 \cdot s_1 \cdot \ldots \\
 & & \exists i[i \geq 0 \wedge s_i \vDash \phi \wedge \forall j[0 \leq j < i \Rightarrow s_j \vDash \psi]] \\
 & & \\
(S,R,s_0) \vDash A(\psi \cup \phi) & \text{iff} & \text{for all path } \tau = s_0 \cdot s_1 \cdot \ldots \\
 & & \exists i[i \geq 0 \wedge s_i \vDash \phi \wedge \forall j[0 \leq j < i \Rightarrow s_j \vDash \psi]] \\
 & & \\
(S,R,s_0) \vDash \psi \rightsquigarrow \phi & \text{iff} & \text{for all path } \tau = s_0 \cdot s_1 \cdot \ldots \\
 & & \forall i[(i \geq 0 \wedge s_i \vDash \psi) \Rightarrow \exists j[i < j \wedge s_j \vDash \phi]]
\end{array}
$$

Before defining our model-checking algorithm, we give some simple results from graph theory that will be useful to prove the soundness of our approach.

5 Some Graph Theoretical Properties

Our model-checking algorithm is based on an iterative exploration of the state space graph. The goal is to prove that a given invariant is valid for every infinite paths in the state space. Since we work with finite state systems, any infinite path includes at least one cycle. Hence, in the remainder of this chapter, we will often focus our attention on the problem of identifying a cycle in a Kripke structure. To this end, we need to define some properties of Directed Acyclic Graphs (DAG).

To give an example; to check the property $A\Diamond(\phi)$ on a Kripke structure KS, it is enough to generate the subset KS_ϕ of KS obtained by "stopping" our exploration whenever we find a state s where ϕ holds. Then, the property is true if KS_ϕ is a DAG; otherwise there would be a cycle of states in KS where ϕ never holds.

Definition 5.1. 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 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 5.1. *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.*

Proof. The proof is by contradiction on the definition of a maximal path in G. We say that $\tau = v_1 \cdot \ldots \cdot v_n$ is a proper path in G if all the vertices in τ are different. We say that τ is a maximal path if it is a proper path and if there are no proper path of size $n+1$. If the in-degree of v_1 is not equal to zero, then there exists a vertex $w \in V$ such that $(w,v_1) \in E$. If w is a vertex in τ then we have found a cycle, which contradicts the fact that G is a DAG. Otherwise, $w \cdot \tau$ is a proper path of G of size $n+1$, which contradicts the fact that τ is maximal. Following a similar reasoning, we can show that the out-degree of v_n is necessarily 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 5.2.

Theorem 5.2. *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.*

Proof. The property is trivial if G is the empty graph.

Otherwise, assume G_0 is a finite DAG with $n+1$ vertices. By lemma 5.1, we know that there is at least one vertex l_0 in G_0 that is a leaf. If we remove this vertex and its incoming edges from G_0, the resulting graph G_1 is a finite DAG with n vertices (removing a vertex cannot introduce a cycle). We can repeat this operation to obtain a sequence $(G_i)_{i\leq n}$ of finite DAG with less and less vertices. Therefore, the graph G_n has only one vertex and this vertex is a root. Actually, there is an index $k \leq n$, that is the smallest value such that all the vertex in G_k have in-degree zero.

To prove the other direction, we assume that we have a finite sequence of graphs $G_i(V_i, E_i)$, with $i \in 0..n$, such that $G_0 = G$, all the vertices in G_n are roots, and we obtain the graph G_{i+1} by removing one leave from G_i. Let l_i denotes the vertex removed from G_i. The rest of the proof is by contradiction. Assume we have a cycle in G, that is to say, there is a path $\tau = v_0 \cdot \ldots \cdot v_{m+1}$ such that $(v_i, v_{i+1}) \in E$ for all index i in $0..m$ and $v_0 = v_{m+1}$. Let v_j be the first vertex in τ to be erased from the graph; that is, there is an index $k \in 0..n$ such that $v_j = l_k$ and all the other states in τ are in G_{k+1}. On one hand, the vertex v_j is well-defined. Indeed, if no vertex from τ is ever erased, then the path τ is also a path in G_n, which contradicts the fact that all the vertices in G_n have in-degree zero. On the other hand, since v_j is a leaf in G_k, this contradicts the fact that there is an edge from v_j to one of the vertex in τ. In consequence, the graph G as no cycles. □

To conclude this section, we study properties of Parental Graphs, that is a spanning subgraph such that all the nodes, except the leaves, have an out-degree of one.

Definition 5.2. 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 a leaf in G then v has an out-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 out from every vertex in G and delete the others. Also, if G is acyclic, then all its parental graphs are acyclic. (Actually, in this case, the parental graph is a spanning tree of the reverse graph.)

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 the following sections, we will 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 5.3. *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 5.1, 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. □

A corollary of this property is that the relation between G and PG is "stable" when we remove the same leaf from both graphs. This property ensures that it is sound to use a parental graph when we recursively remove all the leaves from a graph

Corollary. *Assume v is a zero out-degree node in G; when we remove the vertex v from PG, we obtain a parental graph of the graph obtained after removing v from G.*

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

Definition 5.3. 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}.

6 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 4 and list, for each formula, whether they involve a backward step.

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

Figure 4: List of Supported Formulas.

This algorithm is not very original. We already said that this is, basically, the semantic approach initially proposed by Clarke and Emerson [CE82, Cla99] for CTL model-checking. The same remark applies to the computation of the "fixed point" in parallel. Our main contribution, from the algorithmic viewpoint, is the definition of a version of this algorithm based on the reverse parental graph. 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. 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.

From the interpretation of formulas listed in figure 4, 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.

6.1 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 for the directed graph (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 $\mathrm{suc}(s)$, that record the out-degree of s in KS. The value of $\mathrm{suc}(s)$ 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 $\mathrm{suc}(s).\mathrm{dec}()$ to decrement the value of the label suc for the state s in KS, and the expression $\mathrm{suc}(s).\mathrm{set}(i)$ 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 PKS. 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 $\mathrm{sons}(s).\mathrm{inc}()$ in our pseudo-code. We will decrement the value of $\mathrm{sons}(s)$ during the backward traversal phase.

6.2 Model-checking Reachability properties — $E(\psi \cup \phi)$

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.

We give the pseudo-code for checking the formula $E(\psi \cup \phi)$ in Listing 1. The inputs are the atomic properties ψ and ϕ and the initial state (or vertex) s_0. The algorithm uses a stack, W, to store the "working states" (that corresponds to paths that still need to be explored) and a set, S, to store the states that have already been visited. The algorithm returns true as soon as the property ϕ is found, otherwise, it returns false.

```
1  function BOOL check_e(ψ : pred, φ : pred, s0 : state)
2     Set S ← new Set(∅) ;
3     Stack W ← new Stack(s0) ;
4     while (W is not empty) do
5        s ← W.pop() ;
6        if (s ⊨ φ) then
7           return TRUE
8        elsif (s ⊨ ψ) then
9           forall s' successor of s in KS do
10             if (s' ∉ S) then
11                // s' is a new state
12                S ← S ∪ {s'} ;
13                W.push(s')
14             endif
15          endfor
16       endif
17    endwhile ;
18    return FALSE
```

Listing 1: Algorithm for the formula $E(\psi \cup \phi)$

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

6.3 Model-Checking Liveness Properties — $A(\psi \cup \phi)$

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 5.2.

To give an example of how this algorithm works, we can consider the case of the formula $A\Diamond(\phi)$ (that is $A(\text{True} \cup \phi)$). In the first step, we stop exploring a path whenever we find a state where ϕ holds. The forward exploration ends when all possible paths have been explored and none of them violates the constraints, i.e., ϕ eventually holds for all paths. Then we start removing the nodes with out-degree zero from the graph. The first "leaves" in the graph are necessarily states where the predicate ϕ holds. Next, we also remove states that have had all their successors removed. We can give a simple explanation of the validity of this method. Indeed, at each step we remove states belonging to the set X defined by the following recursive equation: a state s is in X if either (1) $s \vDash \phi$, or (2) all the successors of s are in X. Basically, we compute the semantics of the modal μ-calculus formula $\mu X.(\phi \vee [\text{True}]X)$, that is equivalent to the LRL formula $A\Diamond(\phi)$.

We give the pseudo-code for checking the formula $A(\psi \cup \phi)$ in Listing 2. Like in the previous case, 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

```
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 2: Algorithm for the formula $A(\psi \cup \phi)$

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 3. 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.

The function forward_check_a basically performs the same operations than the function check_e of Section 6.2, with some minor differences. More precisely, the function does not return when we find a state where ϕ holds. Instead, we continue our exploration on other paths of the state graph. On the opposite, we stop the exploration and return false whenever we find a state where both ϕ and ψ do not hold or a dead state where ψ holds.

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[5], 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.

As an example, we show the result of a backward exploration for two Kripke structures, G and G_c in Figure 5 and 6. The graph G_c is obtained by adding an edge to G (pictured in red) that results in the presence of a cycle. In each figure, we show: (1) the graph; (2) the resulting reverse graph with the initial labeling of each node; and (3) the result of the clearing phase. In the last case, a red cross means that the state is cleared, while a simple mark is used for states that have been updated.

Figure 5 gives an example of a successful backward traversal, while figure 6 gives an example of an unsuccessful clearing phase.

Listing 4 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 $\mathrm{suc}(s)$ 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.

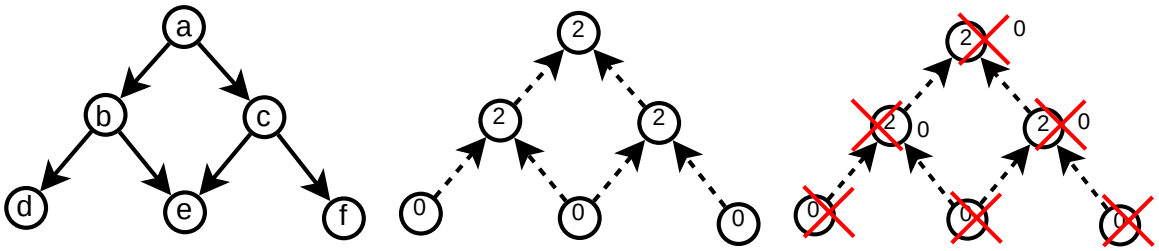
[5] We assume that the decrementing operations are done in parallel.

```
1  function BOOL forward_check_a(ψ : pred, φ : pred, s0 : state, A : Stack)
2     Set S ← new Set(∅) ;
3     Stack W ← new Stack(s0) ;
4     while (W is not empty) do
5        s ← W.pop();
6        if (s ⊨ φ) then
7           // we clear state s from KS
8           suc(s).set(0) ;
9           A.push(s)
10       elsif (s ⊨ ψ) then
11          // we tag s with its number of successors
12          suc(s).set(number of successors of s in KS) ;
13          // check if s is not a dead state
14          if (suc(s) = 0)
15             return FALSE
16          endif
17          // and continue the exploration
18          forall s' successor of s in KS do
19             if (s' ∉ S) then
20                // s' is a new state
21                S ← S ∪ {s'} ;
22                W.push(s')
23             endif
24          endfor ;
25       else return FALSE
26       endif
27    endwhile ;
28    return TRUE
```

Listing 3: Forward exploration for the formula $A(\psi \cup \phi)$ with Reverse Graph

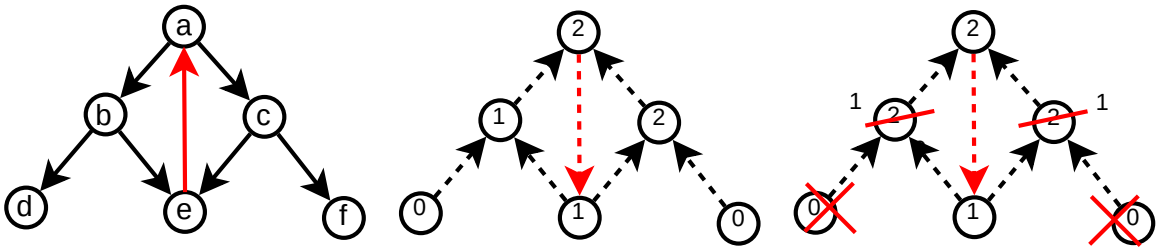
```
1  function BOOL backward_check_a(s0 : state, A : Stack)
2     while (A is not empty) do
3        s ← A.pop() ;
4        // the property is true if we reach the initial state
5        if (s = s0) then
6           return TRUE
7        endif
8        // otherwise we check if the predecessors of s can be cleared
9        forall s' parent of s in KS do
10          suc(s').dec() ;
11          if (suc(s') = 0) then
12             A.push(s')
13          endif
14       endfor
15    endwhile ;
16    return FALSE
```

Listing 4: Backward exploration for the formula $A(\psi \cup \phi)$ with Reverse Graph



Graph G. RG from G. RG backward traversal.

Figure 5: Successful Reverse Graph backward traversal for $A(\psi \cup \phi)$.



Graph G_c with a cycle. RG_c from G_c. RG_c backward traversal.

Figure 6: Unsuccessful Reverse Graph backward traversal for $A(\psi \cup \phi)$.

Algorithm for the reverse parental graph version — RPG

We give the pseudo-code for the forward exploration function, forward_check_a, in the case of the parental graph version (see Listing 5).

The 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. But the algorithm could work with any other choice.

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 6. The algorithm iterates between two behaviors, *clearing* and *collecting*. The *clearing* behavior is similar to the pseudo-code for the RG algorithm (see Listing 4), 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 to check whether they have been cleared also (if their suc label is zero).

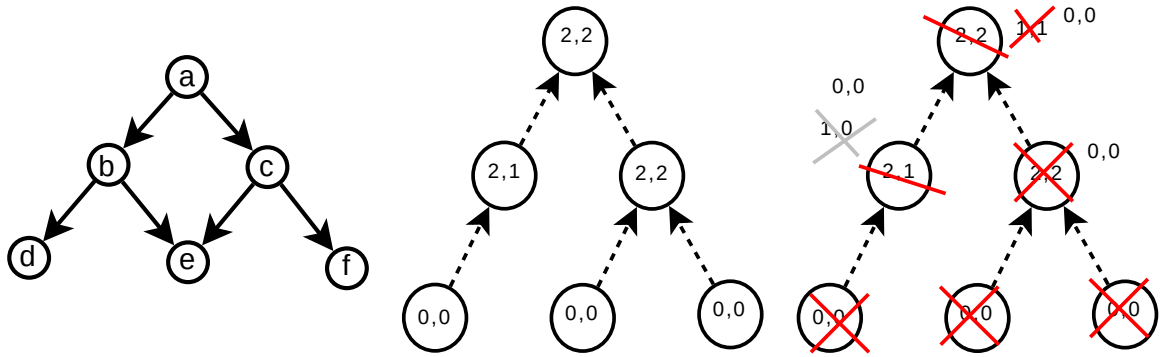
The advantage of this strategy is that we do not have to consider all the states in the graph, just a subset of it. Indeed, we know from Theorem 5.3 that this subset is enough to test the presence of a cycle. At the opposite, the drawback of this approach is that we may try to clear the same vertex several times, which may be time consuming.

We show the result of the backward propagation phase for two different cases in figures 7 and 8. In each diagram, the vertices are decorated with the pair of labels $(\text{suc}, \text{sons})$.

Figure 7 gives an example of a successful backward traversal. Initially, the stack A contains the states

```
1  function BOOL forward_check_a(ψ : pred, φ : pred, s0 : state, A : Stack)
2     Set S ← new Set(∅) ;
3     Stack W ← new Stack(s0) ;
4     while (W is not empty) do
5        s ← W.pop() ;
6        if (s ⊨ φ) then
7           suc(s).set(0) ;
8           A.push(s)
9        elsif (s ⊨ ψ) then
10          suc(s).set(number of successors of s in KS) ;
11          // check if s is not a dead state
12          if (suc(s) = 0)
13             return FALSE
14          endif
15          forall s' successor of s in KS do
16             if (s' ∉ S)
17                // then s is the father of s' PKS
18                S ← S ∪ {s'} ;
19                father(s').set(s) ;
20                sons(s).inc();
21                W.push(s')
22             endif
23          endfor ;
24       else return FALSE
25       endif
26    endwhile ;
27    return TRUE
```

Listing 5: Forward exploration for the formula $A(\psi \cup \phi)$ with Reverse Parental Graph



Graph G. *RPG* from G. *RPG* backward traversal.

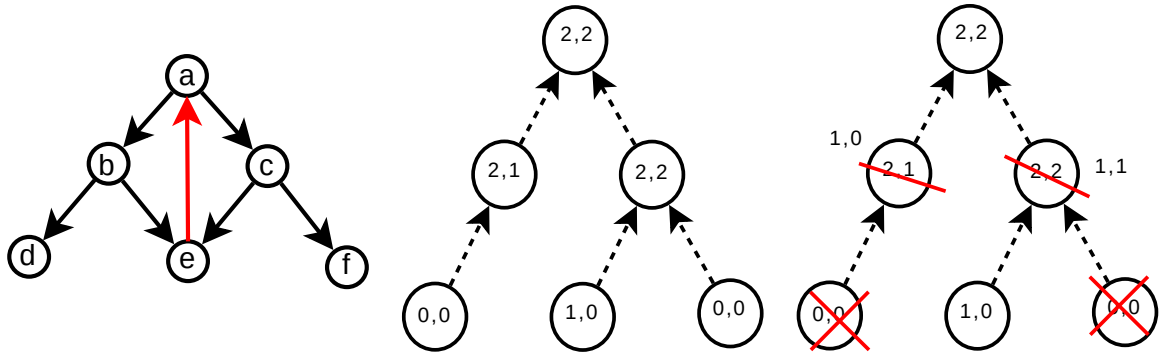
Figure 7: Successful Reverse Parental Graph backward traversal for $A(\psi \cup \phi)$.

d, e, f. After they are removed, we have $\text{sons}(b) = 0$, $\text{suc}(b) = 1$ and $\succ(c) = \text{sons}(c) = 0$. Then state c can be cleared. At this point, we have only one zero in-degree node, b, in the reverse parental graph. We need to recompute one transition to retrieve the fact that e is a successor of b. Since e was already removed, we can finally clear b and then reach the initial state a.

We give an example of unsuccessful backward traversal in Figure 8. Unlike the previous case, when the backward propagation ends, we have two zero in-degree node in the parental graph, namely b and e. Since $\text{suc}(b) = \text{suc}(e) = 1$, we cannot clear any of these states. Therefore the process ends and, by Theorem 5.3, we know that there must be a cycle in G_c.

```
1  function BOOL backward_check_a(ψ : pred, ϕ : pred, s0 : state, A : Stack)
2     over ← FALSE
3     while (not over)
4        while (A is not empty) do
5           //Clearing
6           s ← A.pop() ;
7           // the property is true if we reach the initial state
8           if (s = s0) then
9              return TRUE
10          endif ;
11          // otherwise we check if the father of s can be cleared
12          s' ← father(s) ;
13          sons(s').dec() ;
14          suc(s').dec() ;
15          if (suc(s') = 0) then
16             A.push(s')
17          endif
18       endwhile
19       //Collecting
20       // if we have no more states to clear in A we try to find
21       // candidates among the states with no children in PKS
22       forall s such that sons(s) = 0 and suc(s) ≠ 0 in KS do
23          if test(s) then
24             suc(s).set(0) ;
25             A.push(s)
26          endif
27       endforall
28       if (A is empty) then
29          //No good candidate was found, end backward search
30          over ← TRUE
31       endif
32    endwhile ;
33    return FALSE
34
35 function BOOL test(s : state)
36    forall s' successor of s in KS do
37       if suc(s') ≠ 0 then
38          // at least one successor is not cleared
39          return FALSE
40       endif
41    endfor
42    return TRUE
```

Listing 6: Backward exploration for $A(\psi \cup \phi)$ with Reverse Parent Graph



Graph G_c with a cycle. RPG_c from G_c. RPG_c backward traversal.

Figure 8: Unsuccessful Reverse Parental Graph backward traversal for $A(\psi \cup \phi)$.

6.4 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 9 gives an example of graph for which the formula is valid.

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 hold.

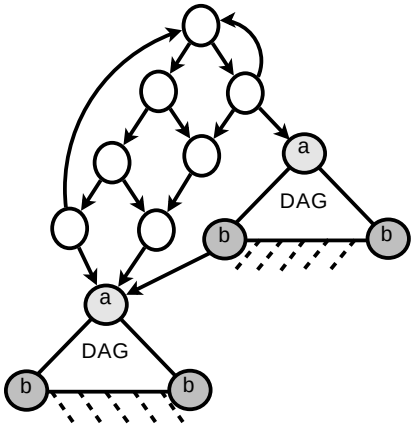


Figure 9: Leadsto $a \rightsquigarrow b$ where a is ψ and b is ϕ.

We give the pseudo-code for checking the formula $\psi \rightsquigarrow \phi$ in Listing 7. Compared to the algorithm for liveness, we use an additional parameter, P, that is a stack where we collect all the states such that ψ holds. Another difference is that we need to explore the state graph totally (we say that leadsto is a global property).

```
1  function BOOL check_leadsto(ψ : pred, φ : pred, s0 : state)
2     Stack A ← new Stack(∅) ;
3     Stack P ← new Stack(∅) ;
4     forward_check_leadsto(ψ , φ, s0, A, P) ;
5     return backward_check_leadsto(A, P)
```

Listing 7: Algorithm for the formula $\psi \rightsquigarrow \phi$

```
1  function void forward_check_leadsto(ψ : pred, φ : pred,
2                                      s0 : state, A : Stack, P : Stack)
3     Set S ← new Set(∅) ;
4     Stack W ← new Stack(s0) ;
5     while (W is not empty) do
6        s ← W.pop();
7        if (s ⊨ φ) then
8           suc(s).set(0) ;
9           A.push(s)
10       else
11          suc(s).set(number of successors of s in KS) ;
12          if (s ⊨ ψ) then
13             P.push(s)
14          endif ;
15          forall s' successor of s in KS do
16             if (s' ∉ S) then
17                S ← S ∪ {s'} ;
18                W.push(s')
19             endif
20          endfor ;
21       endif
22    endwhile ;
23    return void
```

Listing 8: Forward exploration for the formula $\psi \rightsquigarrow \phi$ with Reverse Graph

Algorithm for the reverse graph version — RG

We give the pseudo-code for the function forward_check_leadsto in Listing 8. The function adds to the stack A (resp. P) all the state where ϕ (resp. ψ) holds. During the forward exploration, we also set the states in A as cleared (we set the label suc to zero) because they will be the starting points for our backward traversal.

We give the pseudo-code for the backward traversal in Listing 9. Again, the code is similar to the backward traversal for the liveness case. The main difference is in the termination condition: the function returns true if all the states in P have been labeled as cleared.

Algorithm for the reverse parental graph version — RPG

The algorithm for forward and backward analysis with the Reverse Parental Graph are similar to the previous cases.

Backward traversal (see figure 11) starts from the *accepted* vertices and terminates when there is no more vertex to clear (set its out-degree to zero). It returns true if all the *seeds* vertices are cleared. Its implementation is similar to the pseudo-code presented at figure 6, the algorithm iterates between two phases: *clearing* and *collecting*.

```
1  function BOOL backward_check_leadsto(A : Stack, P : Stack)
2     while (A is not empty) do
3        s ← A.pop() ;
4        forall s' parent of s in KS do
5           suc(s').dec() ;
6           if (suc(s') = 0) then
7              A.push(s')
8           endif
9        endfor
10    endwhile ;
11    while (P is not empty) do
12       s ← P.pop() ;
13       if (suc(s) ≠ 0) then
14          return FALSE
15       endif
16    endwhile ;
17    return TRUE
```

Listing 9: Backward exploration for $\psi \rightsquigarrow \phi$ with Reverse Graph

```
1  function void forward_check_leadsto(ψ : pred, φ : pred,
2                                      s0 : state, A : Stack, P : Stack)
3     Set S ← new Set(∅) ;
4     Stack W ← new Stack(s0) ;
5     while (W is not empty) do
6        s ← W.pop();
7        if (s ⊨ φ) then
8           A.push(s) ;
9           suc(s).set(0)
10       else
11          suc(s).set(number of successors of s in KS) ;
12          if (s ⊨ ψ) then
13             P.push(s)
14          endif ;
15          forall s' successor of s in KS do
16             if (s' ∉ S) then
17                S ← S ∪ {s'} ;
18                father(s').set(s) ;
19                sons(s).inc() ;
20                W.push(s')
21             endif
22          endfor
23       endif
24    endwhile ;
25    return void
```

Listing 10: Forward exploration for $\psi \rightsquigarrow \phi$ with Reverse Parent Graph

```
1  function BOOL backward_check_leadsto(ψ : pred, φ : pred,
2                                       s0 : state, A : Stack, P : Stack)
3     over ← FALSE
4     while (not over)
5        while (A is not empty) do
6           //Clearing
7           s ← A.pop() ;
8           // we check if the father of s can be cleared
9           s' ← father(s) ;
10          sons(s').dec() ;
11          suc(s').dec() ;
12          if (suc(s') = 0) then
13             A.push(s')
14          endif
15       endwhile
16       //Collecting
17       // if we have no more states to clear in A we try to find
18       // candidates among the states with no children in PKS
19       forall s such that sons(s) = 0 and suc(s) ≠ 0 in KS do
20          if test(s) then
21             suc(s).set(0) ;
22             A.push(s)
23          endif
24       endforall
25       if (A is empty) then
26          //No good candidate was found, end backward search
27          over ← TRUE
28       endif
29    endwhile ;
30    // the property is true if all the state in P are cleared
31    while (P is not empty) do
32       s ← A.pop() ;
33       if (suc(s) ≠ 0) then
34          return FALSE
35       endif ;
36    endwhile ;
37    return TRUE
38
39 function BOOL test(s : state)
40    forall s' successor of s in KS do
41       if suc(s') ≠ 0 then
42          // at least one successor is not cleared
43          return FALSE
44       endif
45    endfor
46    return TRUE
```

Listing 11: Backward exploration for $\psi \rightsquigarrow \phi$ with Reverse Parent Graph

7 Correctness and Complexity of our Algorithms

In this section, we give a correctness proof for the MCLCD algorithm. We also study the complexity of our algorithm in the sequential case. We more specifically study the case of the liveness formula $A(\psi \cup \phi)$. The results obtained for this case can be generalized to our whole logic.

The correctness of our algorithm is based on the fact that the computation will stop (and return the boolean value FALSE) if there is at least one cycle in KS, that is to say, we cannot perform a complete backward traversal and reach the initial state.

Theorem 7.1 (Termination)**.** *The MCLCD algorithm, for model-checking the logic LRL on a finite Kripke Structure, terminates for all inputs.*

Proof. We only consider the case for the formula $A(\psi \cup \phi)$, that is the function check_a. The other cases are similar. We prove the termination of check_a by proving the termination of the two functions that it calls: forward_check_a and backward_check_a.

The function forward_check_a (see for instance Listing 3 for the RG case) generates a subset of the state graph by pruning all the transitions after a state where ϕ holds. At each iteration of the function, we consider a different state taken from a stack—denoted S in the pseudo-code—that contains a subset of the reachable states. Since the state graph is finite, the function will always terminate.

For the function backward_check_a, termination follows from the fact that, at each iteration, we decrement the value of at least one of the labels $\mathrm{suc}(s)$. □

Theorem 7.2 (Completeness)**.** *If the system KS satisfies the formula F then the MCLCD algorithm returns the boolean value TRUE when run with the value* (KS, F).

Proof. We only consider the case for the formula $A(\psi \cup \phi)$. The proof is essentially the same for the RG and RPG cases. (The two algorithms have a similar behavior; only their complexity differ.) We assume that $KS \vDash F$ and study the result of the expression check_a(ψ, ϕ, s_0).

By the definition given in Section 4, we have $KS \vDash A(\psi \cup \phi)$ if and only if (INV): for all maximal path $\tau = s_0 \cdot s_1 \cdot \ldots$ in KS there is an index i such that $s_i \vDash \phi \wedge \forall j[0 \leq j < i \Rightarrow s_j \vDash \psi]$.

First, we prove that the call to forward_check_a(ψ, ϕ, s_0, A) returns TRUE. The proof is by contradiction. Let us assume that the forward exploration returns FALSE (we know that the forward exploration terminates). Hence, there must be a path $\tau = (s_i)_{i \in 1..n}$ such that $s_i \vDash \psi \wedge \neg\phi$ for all $i < n$ and $s_n \vDash \neg\psi \wedge \neg\phi$. Since we can always extend this path into a maximal path of KS, this contradicts (INV). Therefore, the forward exploration must succeeds.

In the remainder of this proof, we assume that KS_f denotes the subset of the state graph generated during the forward exploration and that A_{init} is the set of states s such that $\mathrm{suc}(s) = 0$. It is exactly the states with out-degree zero in KS_f and the set of states in KS_f where ϕ holds. By construction, since at least one state must satisfy ϕ, the set A_{init} is not empty. Also, the parameter A is set to A_{init} when we start the backward exploration phase.

Now, we consider the backward exploration for the reverse graph case. Let A_i denotes the set of states that are cleared in KS_f after the i^{th} iteration of the while loop in the call to backward_check_a. The sequence of sets $(A_i)_{i \in 1..n}$ can be defined by the following induction: (1) $A_0 = A_{init}$, and (2) for all $i > 0$, we have $A_{i+1} = A_i \cup \{s \| \forall s'.s\, R\, s' \Rightarrow s \in A_i\}$. Since it is an increasing sequence, bounded by the finite set S, it finally reaches a limit, $A_f \subseteq S$. If we use the terminology defined in Sect. 5, $S \setminus A_f$ are exactly the states that are left in KS_f when we recursively remove all the cleared states.

To prove that the backward exploration also succeeds, it is enough to show that $KS \vDash A(\psi \cup \phi)$ entails $s_0 \in A_f$. From (INV), we have that KS_f must be a DAG. Otherwise we could find a maximal path in which ϕ never holds. Therefore, by Theorem 5.2, we have $s_0 \in A_f$, as needed. □

Theorem 7.3 (Soundness)**.** *If the MCLCD algorithm returns TRUE for a given pair (KS, F) of a Kripke Structure and a LRL formula then $KS \vDash F$.*

Proof. Once again, we only give the proof in the case where F is the formula $A(\psi \cup \phi)$. The proof is essentially the same for the RG and RPG cases. (The two algorithms have a similar behavior; only their complexity differ.)

Let us assume that the call to check_a(ψ, ϕ, s_0) returns TRUE. Then the forward and backward exploration must both succeeds.

We assume that KS_f is the subset of the state graph computed during the forward exploration. By construction, the predicates ψ holds for all the states in KS_f except for its leaves, where ϕ holds.

Next, we show that if the backward exploration succeeds then KS_f is a DAG. The proof is the same than with Theorem 7.2 and makes use of the other part of the equivalence given by Theorem 5.2

Since KS_f is a DAG, all the maximal paths in KS_f must eventually reach a state where ϕ holds. Therefore, since each maximal path in KS is necessarily an extension of a maximal path in KS_f, the same property is true with KS, which means that $KS \vDash A(\psi \cup \phi)$. □

Next, we study the worst-case complexity of our algorithms. We obtain different results for the RG and RPG versions of the MCLCD algorithm. (Recall that, inside asymptotic notations, we use the symbols S and R when we really means $|S|$ and $|R|$.)

Theorem 7.4 (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.*

Proof. In the worst-case, we need to perform both a forward and a backward exploration.

The complexity of the forward exploration is trivially bounded by the size of the state space: in the worst-case, we need to explore all the states and test all the transitions. Hence the complexity of this first phase is (linear) in $O(S+R)$.

The complexity of the backward exploration depends on the underlying data structures used to encode the Kripke structure. For each version of our algorithm, the worst case is when the property is true.

If we rely on a reverse graph structure—and if we assume that the property is true—then we need to clear all the states in the graph and, for every transition (edge) in the reverse graph, we need to update the label of its head. Therefore, the complexity of the backward exploration is also $O(S+R)$ in the worst case. That is, the overall complexity is in $O(2 \cdot (S+R))$ for the RG version of our algorithm. This result is consistent with the complexity of the CTL model-checking algorithm defined in [CES86], that has a time complexity of $O(|\phi| \cdot (S+R))$, where $|\phi|$ is a measure of the complexity of the specification. Indeed, in our case, the most complex formula (the leadsto property) has complexity 2.

The analysis is quite similar for the version based on the reverse parental graph. The main difference is that we may have to recompute some transitions in KS several time. At least, we need to recompute all the transitions whose "inverse" is not stored in the reverse parental graph.

Since a Kripke structure is a weakly connected graph, we have that $|R| > |S|$ and that $|R| - |S| + 1$ is a bound to the number of transitions not stored in the RPG. At each iteration, we need to test the successors of the states, s, such that $\text{sons}(s) = 0$. This is in order to find the zero out-degree node in the Kripke structure, that is, the states s such that $\text{suc}(s) = 0$. In the worst case, we may have to re-compute all the transitions that are not stored in RPG.

We clear at least one state and remove at least one transition at each iteration of the backward check function. Therefore, there is at most $|S|-1$ iterations and, at the i^{th} iteration, there is at most $|S|-i$ states that are not cleared and $|R|-|S|-i+1$ transitions that may need to be re-computed. Thus, the complexity of our algorithm, for the i^{th} iteration, is bounded by $(|S|-i)+(|R|-|S|-i+1)$. As a consequence, the complexity of the backward traversal can be bounded by the following expression, which is in $O(S\cdot(R-S))$.

$$\begin{aligned}\sum_{i\in 1..|S|-1}(|S|-i)+(|R|-|S|-i+1) &= 1/2.|S|.(|S|-1)+(|R|-3/2.|S|+1).(|S|-1)\\ &= (|R|-|S|+1).(|S|-1)\end{aligned}$$

□

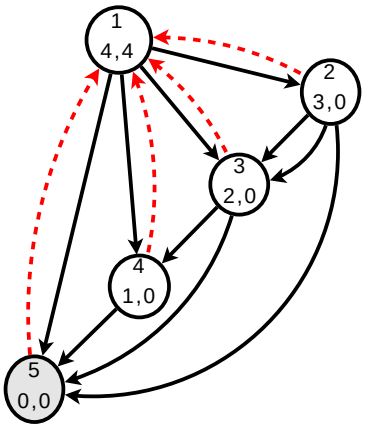


Figure 10: Worst-Case Example for the RPG Version (edges in red are in the reverse parental graph).

Since the number of transitions 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. We can show that this is an asymptotic tight bound for the complexity of the backward exploration.

In the case of the RPG version, we give an example of state graphs such that the complexity is asymptotically equivalent to S^3 (up to a constant). In Fig. 10, we give an example of a state graph, with 5 states, that fall in the "worst complexity" case. Following the same structure than in this example, we can build a family of graphs K_N with N states and $1/2.N.(N-1)$ transitions (for any $N>2$); K_N is a directed graph with N vertices that has the maximum possible number of edges for a DAG.

The backward exploration on K_N will require $N-1$ iterations. Actually, when we remove the only leave in the graph K_N we obtain the graph K_{N-1}. A more precise analysis of the behavior of the backward exploration of the graph K_N gives a complexity that is a factor of $N^3+O(N)$; if we denote $C(N)$ the complexity for the graph K_N then we have the relation $C(N+1)=1/2.N.(N-1)+N+C(N)$.

In conclusion, we have shown 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. Note that this is a complexity result for the worst-case, obtained using a family of graphs with unbounded degree. (The maximal degree of graph K_N is $N-1$.)

In particular, we can expect a smaller difference in time complexity if we consider that the out-degree of the state space is bounded.

8 Parallel Implementation of our Algorithm

While we consider a Random Access Machine (RAM) model for the complexity results given in the previous section, we have not specifically fixed the abstract computational model that is used to interpret the semantics of our pseudo-code. Most particularly, 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 [TSDZB11].

In a SPMD context, all processing units will execute the same functions, as defined in Sect. 6. 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 introduced in [TSDZB11]) 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 $\mathrm{sons}(s).\mathrm{dec}()$). 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 5.3 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 [TSDZB11]) 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.

9 Experimental Results

We have implemented several versions of our model-checking algorithm as part of our prototype model checker MERCURY ([Saa11]). They are built on top of our previous algorithm for parallel state space exploration (see [TSDZB11]). We basically follow the same guidelines than in our implementation of state space generation: we use the C language with Pthreads [But97] for concurrency and the Hoard Library [BMBW00] for parallel memory allocation. We use our *Localization Table* to store the set of explored states. 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 used models taken from two sources. We have three classical examples: Dining Philosophers (PH); Flexible Manufacturing System (FMS); and Kanban – taken from [MC99] – together with 5 Puzzles models: Peg-Solitaire (Peg); Sokoban; Hanoi; Sam Lloyd's puzzle (Fifteen); and 2D Toads and Frogs puzzle (Frog) – taken from the BEEM database [Pel07]. All these examples are based on finite state systems modelled using Petri Nets [Mur89]. This means that, in these cases, a state is a *marking*, that is a tuple of integers. Our algorithm may be adapted to other formalisms, for instance including data, time, etc.

Figure 11 lists the formulas and models selected for our experiments. We choose the Dining Philosophers (PH), the Token Ring (TK), the Peg-Solitaire (Peg) and the Sokoban (SK) models in order to have different types of state graphs for the verification.

The PH and TK benchmarks are classical models that are very well-suited for verification methods based on partial order techniques due to their high degree of interleaving and symmetries. The puzzle models (Peg and SK) have an opposite behavior and have graphs structures that "are almost" DAG.

For each model, we list the formulas that have been checked and give an informal description of the specification. In each case, we try to have 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 ($\psi \rightsquigarrow \phi$). Actually, our tool uses an equivalent ASCII syntax: `E<>`, `A<>`, `E[]`, `A[]`, and `==>` for the modalities in LRL and `- phi` for the negation on predicates. We also use a special predicate, `dead`, to denote the states without successors.

For the model TK, we experimented with two different versions: one that is simply called TK, which is the classical Token Ring example, where starvation is possible—a process can be perpetually denied access to the service ("token")—and a second version, that we call TK_M, which is a modified model that avoids the resource starvation problem. Similarly to our previous analysis, all these examples are based on finite state systems modeled using Petri Nets [Mur89].

The rest of the section is divided in two main parts. First, we perform a speedup analysis of our model-checking program. We also try to analyze how our new approach for detecting cycles participate to the total speed-up of the method. Finally, we present a broader comparison of the two solutions proposed in this work with a third variant where we do not store the transitions, that is to say, only the states are stored and the reverse transition relation is are re-computed dynamically during the backward phase.

Model	Formula	Description	Results
Sokoban(SK) $7 \cdot 10^7$ states $18 \cdot 10^7$ trans.	`E<> win`	The Game has a wining move	*true*
	`E[] - win`	There is an infinite match.	*true*
Philosophers (PH) $14 \cdot 10^7$ states $17 \cdot 10^8$ trans.	`E<> gr1 /\ gl1`	A philosopher can eventually eat.	*true*
	`A<> (gr1 /\ gl1)`	A philosopher will eventually eat.	*false*
	`E[] - (gr1 /\ gl1)`	A philosopher may never eat.	*true*
	`A[] - ((gr1 /\ gl1) /\(gr2 /\ gl2))`	Two philosophers cannot both eat at the same time (mutual exclusion).	*true*
	`(wl1 /\ wr1) ==> (gl1 /\ gr1)`	Whenever a philosopher wants to eat, then eventually it will eat (starvation)	*false*
Solitaire (33 pegs) (PEG) $18 \cdot 10^7$ states $15 \cdot 10^8$ trans.	`E<>(peg_1 +... + peg_33 = 1)`	There is a sequence of moves leading to a winning position (only one peg left).	*true*
	`A<> dead`	Every sequence of moves eventually end in a dead position (no more pegs to remove).	*true*
Token Ring (22 stations) (TK/TK_M) $23 \cdot 10^7$ states $22 \cdot 10^8$ trans.	`wait_1 ==> cs_1`	No process should wait indefinitely to enter its critical section.	(TK) *false* (TK_M) *true*
	`A[] -(cs_1 + ... + cs_22 > 1)`	We cannot have more than one process in critical section at any time.	*true*
	`E[] -(cs_1 + ... + cs_22 = 0)`	We can find a scenario where no process enters its critical section.	*true*

Figure 11: Formulas and Models in our Benchmark.

9.1 Speedup Comparison Between the RG and RPG Algorithms

We analyze the speedup of our parallel model checking algorithm for the benchmark described in Figure 11. We give the relative speedup and the execution time for the reverse and reverse parental graph versions our algorithm. 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.

Experimental Results for the Dining Philosophers Model — PH

We give the speedup analysis for the Dining Philosophers (PH) in Figures 12 and 13: we display the speedup and the execution time for different configurations, from 1 to 16 processors, and for both ver-

sions of our algorithm. Each diagram has one line chart for each kind of formulas. The results are fairly good, with an average efficiency of 65% for the reverse version and 56% for the parental version.

We can explain the abnormal behavior for the reachability formula $E <>$ by the fact that the algorithm is very fast in this case—it takes less than one second to finish in average—and therefore the results are prone to experimental fluctuations. (We can use Fig. 26 to compare the execution time for the different formulas.)

Although we could expect the parental algorithm to have a very good speedup—it has more opportunities to benefit from the parallelism—the relative speedup for the RPG version is not as good as the one obtained by the RG version. This surprising result is partially linked to the fact that the sequential execution time (execution time on one processor) is significantly lower for the parental version than for the reverse version. On of the reasons explaining this behavior is that the forward exploration phase of the RPG algorithm is much faster (there is less information to write in memory).

We try to pinpoint more precisely what is the impact of our two different versions on the performance in Figure 14. We give several bar charts where we decompose the speedup into three separate values: the speedup for the forward exploration phase, the speedup for the backward exploration phase (cycle detection), and finally the overall speedup. For a better view of the impact of each phases, we display the "cumulative" execution time in Fig. 26, obtained when checking the PH model with 16 processors.

For the PH model, we can observe that, in both variants, the speedup for the exploration phase is: (1) much more important than the one obtained in the cycle detection phase; and (2) almost equal or slightly better than the total speedup. We remind the reader that there is no cycle detection phase when we model-check the formulas `E<>` or `A[]`.

Concerning the RG algorithm (the top graph from Fig. 14), we observe that the speedup for the cycle detection phase is very poor for the PH model. This can be explained by the fact that this phase is very short (we can see in Fig. 26 that the time spent in this phase is negligible compared to the time spent in the forward exploration) and do not create enough work for several processors; which means that our work stealing strategy has no effect. The results are more interesting for the Peg model, that we study later.

Concerning the RPG algorithm (the lower graph in Fig. 14), we observe better speedups for the cycle detection phase, with values slightly superior to 4 when using 16 processors. We also observe that, for more than 10 processors, the execution time of the RPG and the RG versions are almost identical (see Fig. 12 and 13). Like with the RG version, the speedup of the algorithm for the PH model mostly comes from the forward exploration phase. In particular, we observe that the efficiency of the exploration phase is the key factor for the performance of the algorithm.

Experimental Results for the Peg Solitaire Model — PEG

We performed the same kind of analysis with a model corresponding to the puzzle game *Peg-Solitaire* (see Fig. 15 and 16). This model offers a good benchmark for our method because the state graph of the system is acyclic. In this benchmark, we only consider one specification, that is an instance of liveness property (`A<> dead`).

Like in the previous case, we give the "cumulative" execution time in Fig. 30, obtained when checking the PEG model with 16 processors.

Our results with the PEG model contrast with the ones obtained with the Dining Philosophers. We observe very good speedups for both versions of our algorithm (with an efficiency of 75%). Also, it is interesting to remark that, even though the relative speedups for the RG and RPG versions are similar, there is a significant difference between their execution time.

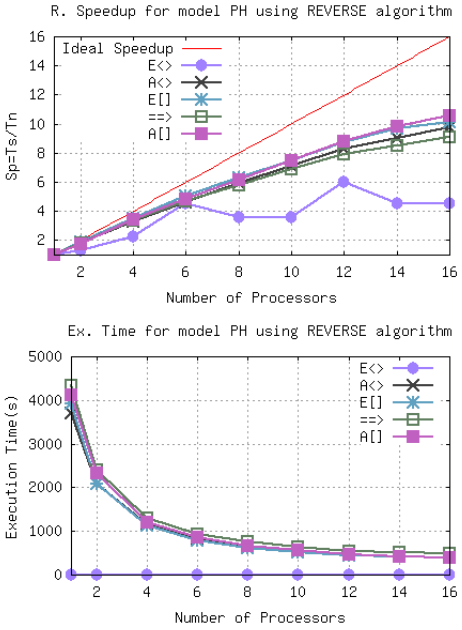


Figure 12: PH with Reverse algorithm.

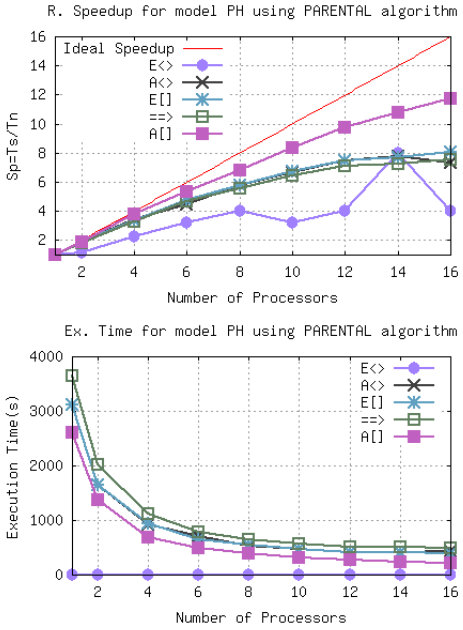


Figure 13: PH with Parental algorithm.

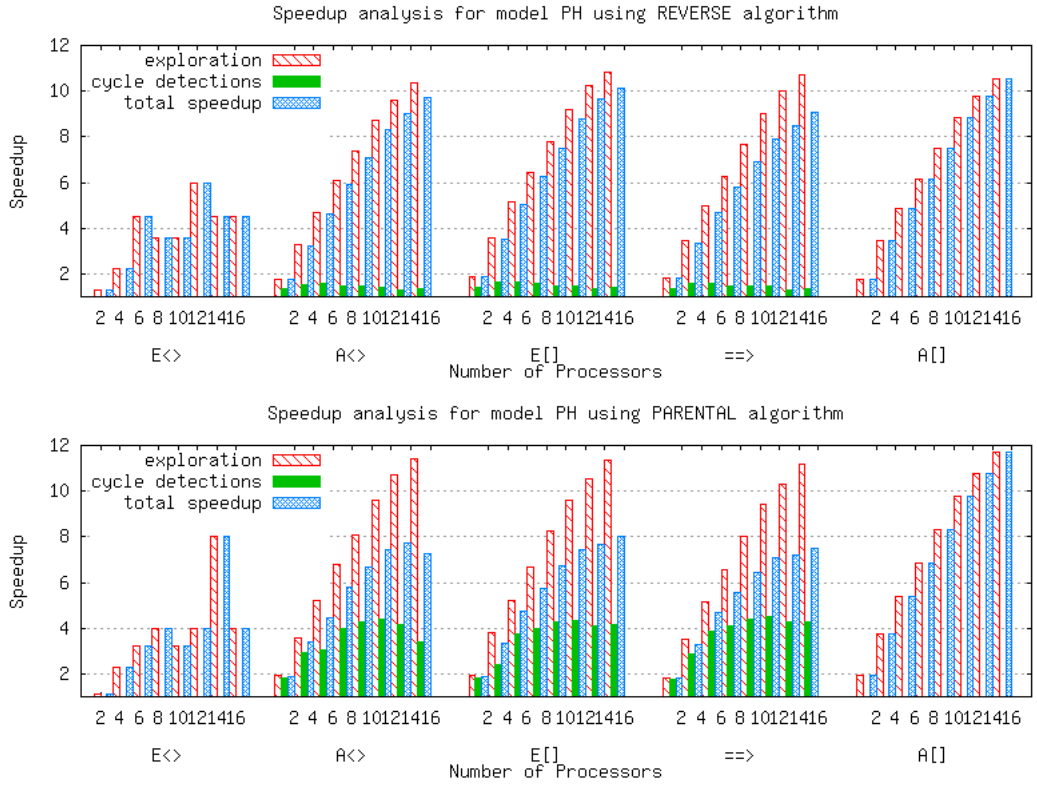


Figure 14: Exploration and cycle detection speedup analysis for PH model.

Figure 17 gives the speedup achieved by the exploration and cycle detections phases independently. We can see that the cycle detection phase has a bigger impact and better speedups. For instance, we have a speedup of approximately 11 for 16 processors for both versions. We can explain this behavior by the fact that, for the formula that we check on the PEG model, we need to completely explore the state graph. It means that the occupancy rate of the processors is better in this example and therefore we take benefit from our work stealing strategy. This is an evidence that our approach is optimized for the worst case scenario when model-checking a system, that is for the case when the property is true.

Experimental Results for the Token Ring Models — TK and TK_M

Finally, we give the results for a model corresponding to the Token Ring protocol. We consider two versions of the protocol. TK stands for the classical "implementation", where starvation is possible. TK_M is a modified version, without starvation, which means that "whenever a process requires a resource, it will eventually be granted the right to use it".

In our benchmark, the most interesting specification is related to starvation, that is an example of the leadsto property: `wait_1 ==> cs_1`. We also consider two examples of safety property: `A[] - (cs_1 + ... + cs_22 > 1)` and `E[] - (cs_1 + ... + cs_22 = 0)`. For these two examples, the first formula do not require a backward exploration phase.

Our results are given in Fig. 18, 19 and 20 for the TK model and Fig. 21, 22 and 23 for the TK_M model.

This example is interesting because we can compare the performance of our algorithms using: state spaces that are very similar; using the same formula; but with a specification that is true in one case and false in the other.

Figures 18 and 19 show the speedup and execution time for the three formulas given in Fig. 11 for the TK model. We observe similar results than with the PH model: the two versions have similar parallel execution time and the total speedup is mainly dominated by the (forward) exploration phase. Like with the PH model, we can explain these results by the fact that the cycle detection phase is very short and stop before completely exploring the state space.

Figures 21 and 22 show the results of a similar experiment for the TK_M model but with a small number of stations, 20 instead of 22. We decided to reduce the number of stations due to the long runtime necessary for the execution with small number of processors, i.e. the sequential time for the parental algorithm is around 12000 seconds for this experiment.. Like we mentioned before, the formula `wait_1 ==> cs_1` is valid for TK_M , that is to say, we need to explore the complete state graph in the backward exploration phase. This explain why, in this case, the results are more similar to the PEG case than to the PH case.

There are some differences though. The execution time of the RPG version is significantly slower and does not scales well for TK_M. We can explain this loss of performance by the number of iterations in the backward exploration. For the PEG model, our algorithm requires 29 iterations and almost one billion transitions are re-computed. for the TK_M model, while the algorithm requires 98 iterations to re-compute $2 \cdot 10^9$ transitions. The difference is quite important, especially since the TK models has four times less states than PEG. (TK_M has $5 \cdot 10^7$ states and $4 \cdot 10^8$ transitions; PEG has $2 \cdot 10^8$ states and $22 \cdot 10^8$ transitions.)

This difference in behavior can be explained by the influence of the state space's "shape" on the algorithm. To give a good idea of what is intended by the notion of "shape" in this context, we display in see Fig. 24 and 25 the state graphs for simplified versions of PEG (only 13 pegs) and TK_M (only 2

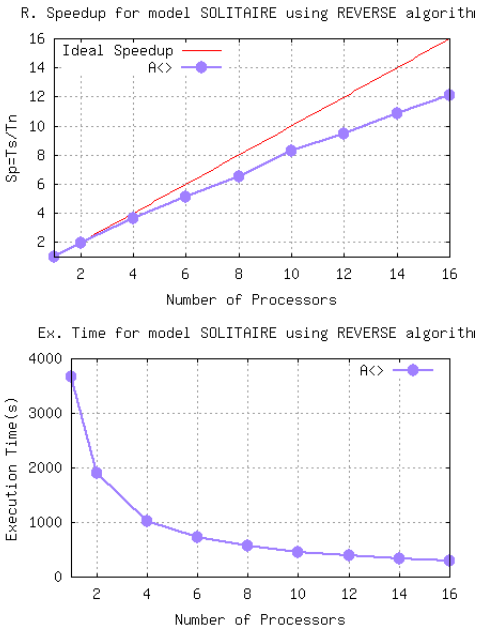


Figure 15: Peg with Reverse alg.

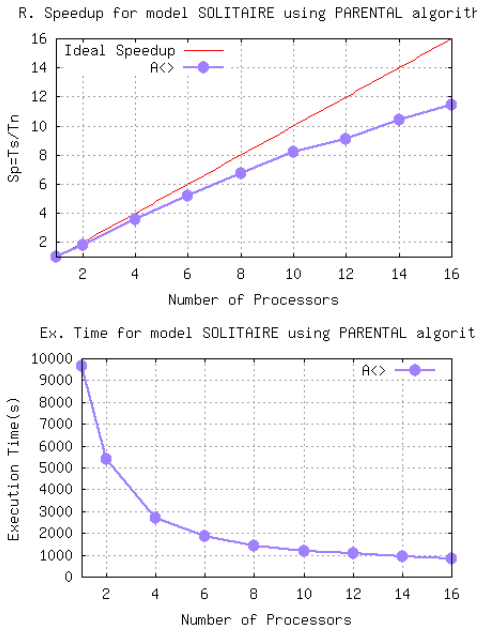


Figure 16: Peg with Parental alg.

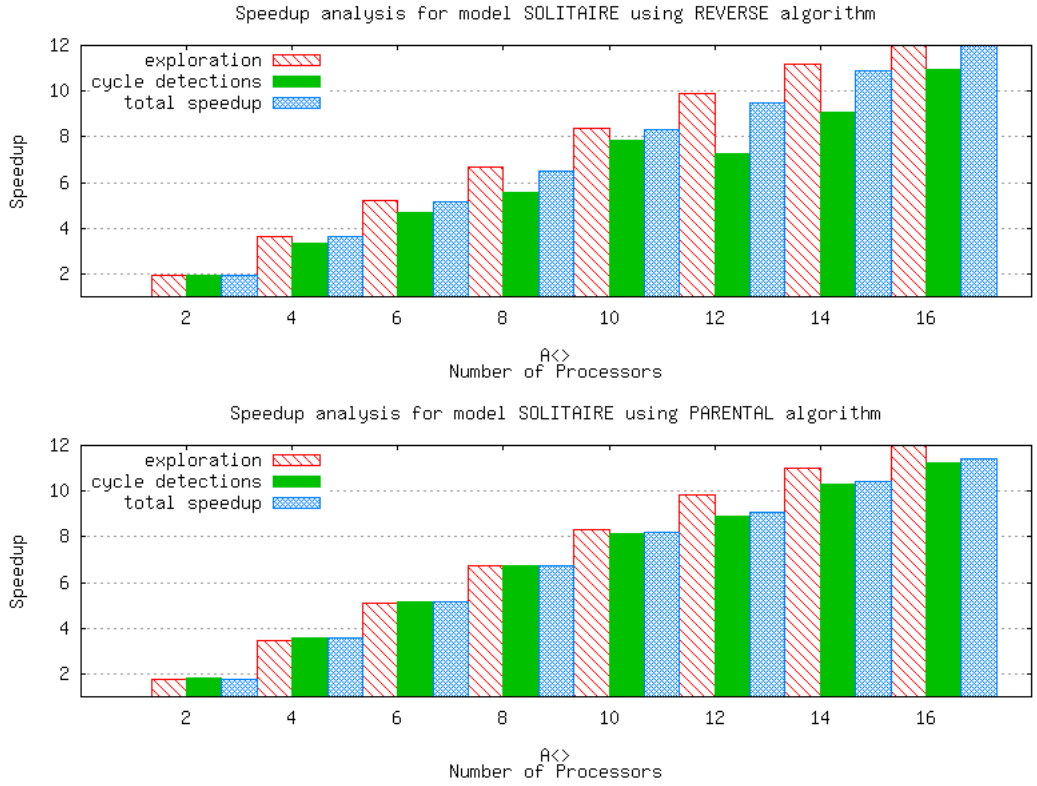


Figure 17: Exploration and cycle detection speedup analysis for Peg model.

stations). Comparing the state graphs for TK_M and PEG, we see that there are less opportunities with TK_M to clear a large number of states in the same iteration.

9.2 Comparison with a Standard Algorithm

In this section, we compare our approach with a "standard" algorithm for model-checking CTL. We assume that we know how to efficiently compute the predecessors of a state in the state graph, that is, that we can directly compute the reverse transition relation. This is true for the models used in our benchmarks, because we know how to easily compute the inverse of a transition in a Petri Net.

In this case, we can simply use the same code than for the RG version of the MCLCD algorithm, 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. The NO_GRAPH version of the algorithm is interesting for several reasons:

- Obviously, it is even more memory efficient than the RPG version. This means that we have the same benefits than the RPG version for the forward exploration, that is, a good speedup due to the fact that we write less information on memory. We also have the same benefits than the RG version for the cycle detection phase, that is, we will never have to re-compute the transitions;
- We can reuse the same data structures and synchronization patterns than in our implementations of the RG and RPG versions. We also use exactly the same models, expressed in the same modelling language. This means that we can really compare algorithms and not only implementations
- Finally, NO_GRAPH is a "standard" algorithm, that is representative of the current state of the art for semantic-based model-checking algorithms.

Next, we give experimental results comparing our implementation of the three versions of the MCLCD algorithm (RG, RPG and NO_GRAPH) using 16 processors on our test machine. Figures 26 to 30 give a series of bar charts where we put in evidence the time required for each phase of the algorithm (exploration and cycle detection).

For the Dining Philosophers (PH) and the Sokoban (SK) models, we observe that: (1) NO_GRAPH has the best execution time; (2) the time spent in the forward phase is the same for NO_GRAPH and RPG; and (3) the RPG algorithm matches the RG algorithm because its gain in performance during the forward exploration exceeds is loss of performance during the cycle detection.

The second observation is not surprising since the forward phase is almost the same for the RPG and NO_GRAPH versions. (RPG should be a little bit slower than NO_GRAPH because we need to store one additional pointer in each state for the father, but this is not noticeable.)

We perform a similar comparison with our two models for the Token Ring protocol (TK and TK_M). As in the previous case, the performances of the versions are essentially the same for the `A[]` and `E[]` formulas. We can even observe that, for the TK model and the `E[]` formula, RPG beats NO_GRAPH. The same is true in the case of the PEG model (see Fig. 30).

The result is far more different for the leadsto formula (`==>`). We remind the reader that, in this case, the formula is false for TK and true for TK_M. We observe that the execution time with RPG are around 7 times slower than with NO_GRAPH.

9.3 Conclusion About the Experiments

We have observed two main categories of behaviors in the analysis of our experimental results. We have examples of *complete backward traversal* and examples of *negligible backward traversal*.

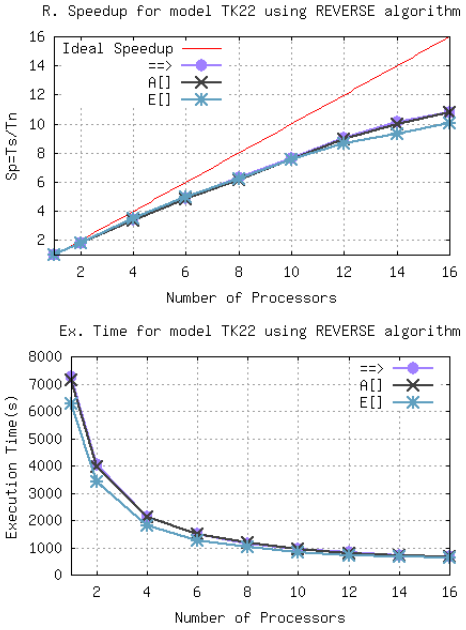


Figure 18: TK with Reverse algorithm.

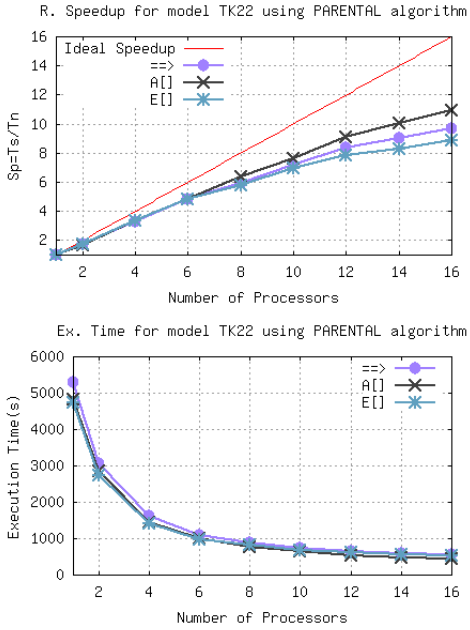


Figure 19: TK with Parental algorithm.

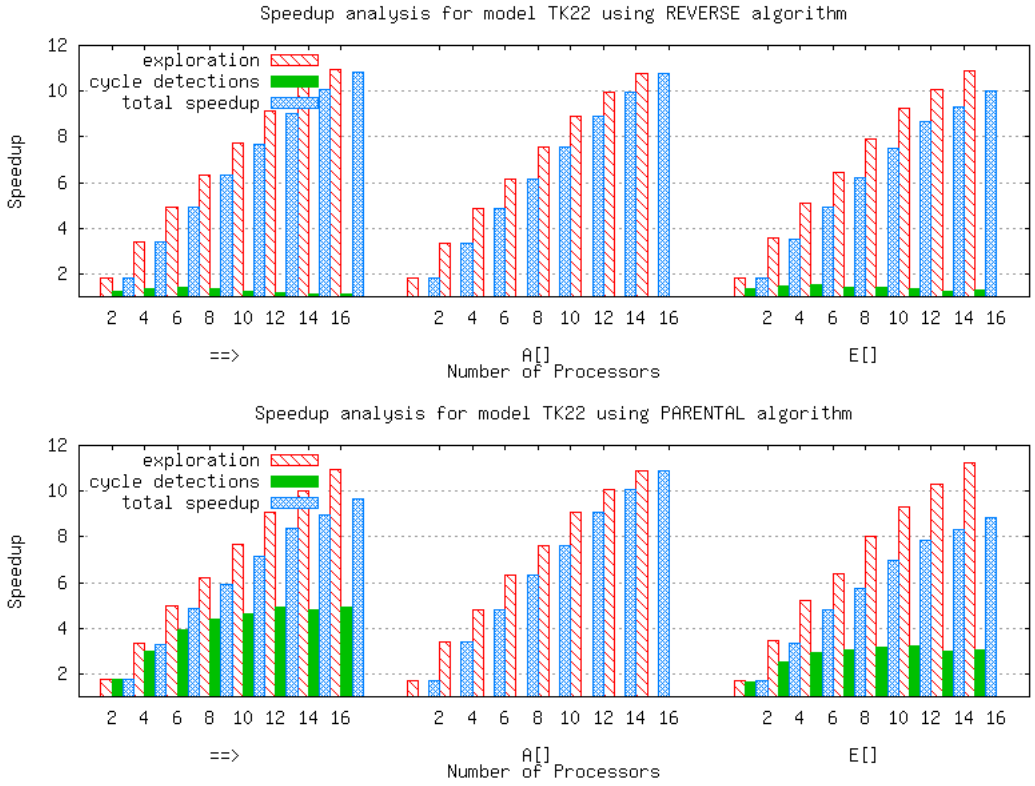


Figure 20: Exploration and cycle detection speedup analysis for TK model.

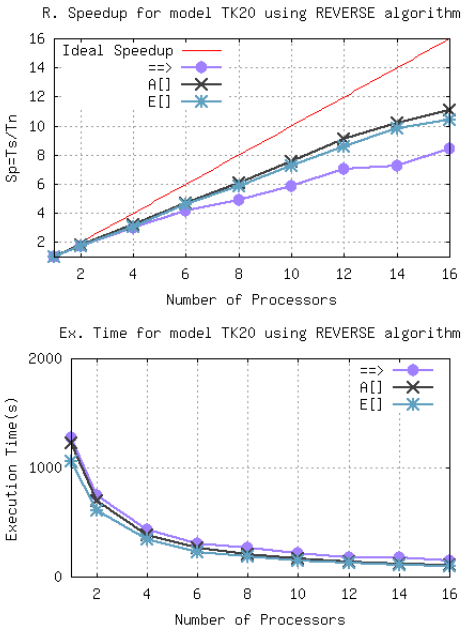


Figure 21: TK_M with Parental alg. .

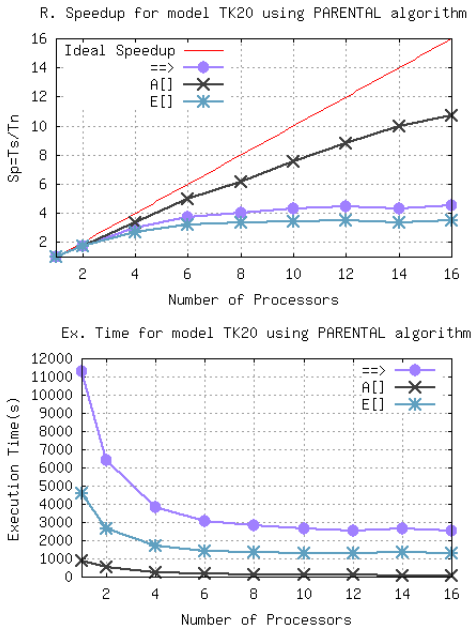


Figure 22: TK_M with Parental alg. .

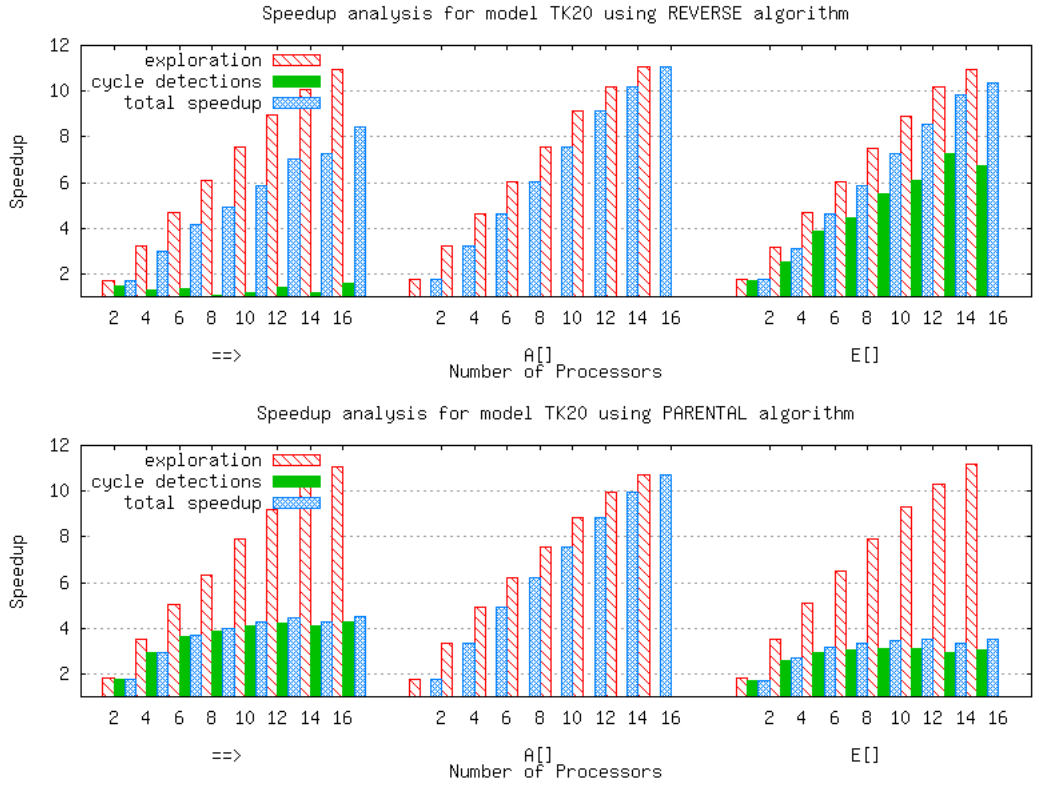


Figure 23: Exploration and cycle detection speedup analysis for TK_M model.

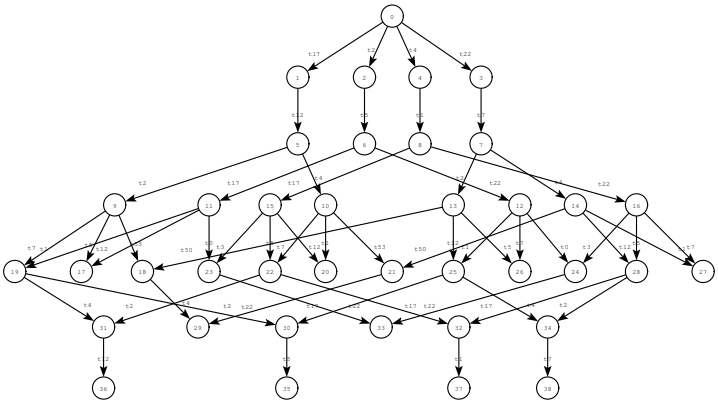


Figure 24: Simplified graph for Peg-Solitaire (13 tokens).

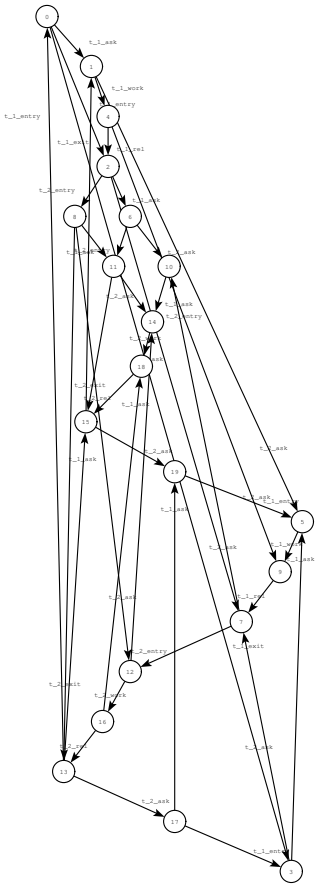


Figure 25: Simplified graph for TK_M (2 stations).

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. 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

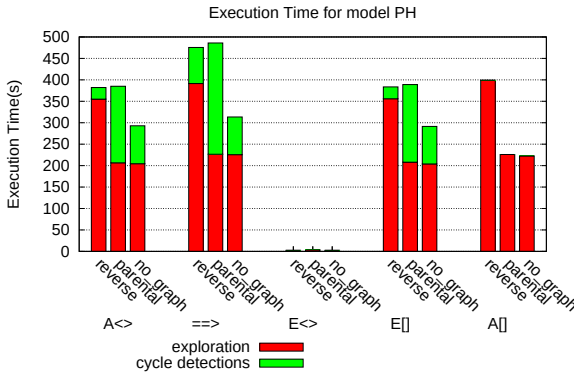


Figure 26: PH model.

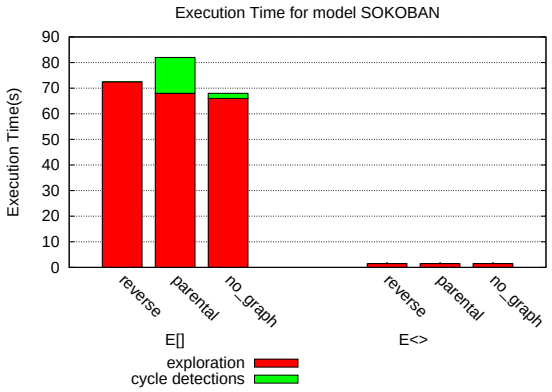


Figure 27: SK model.

Figure 28: TK model.

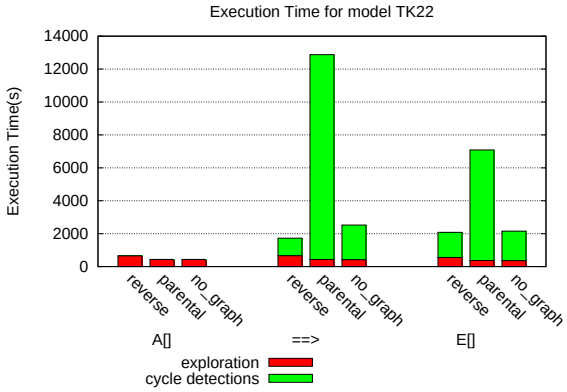


Figure 29: TK_M model.

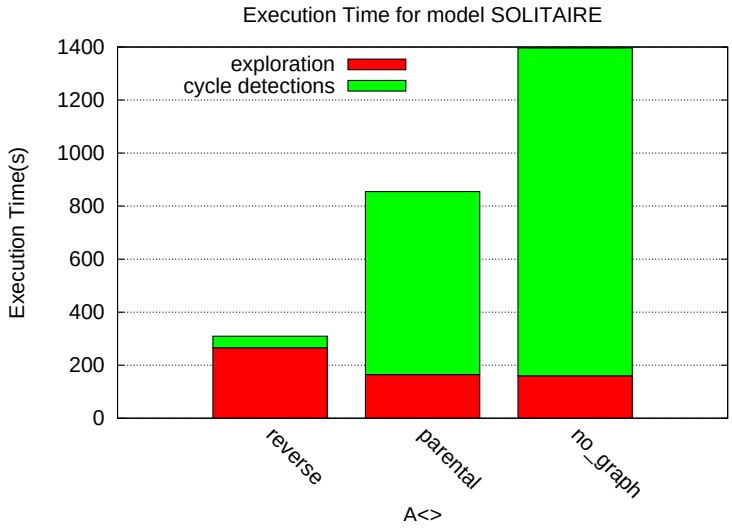


Figure 30: PEG model.

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. 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. We also observed that, in this case, the "shape" of the state graph has a strong impact on the performance of the algorithm; in particular, the RPG algorithm does not scale well when the backward traversal phase requires a lot of iterations between the clearing and collecting steps.

We can draw some conclusions from these benchmarks. The RG version of our algorithm appears to be the best choice when we expect to spend a non-negligible part of the execution time in the cycle detection phase. When the backward phase is short, the NO_GRAPH and RPG versions are a good choice. This is in particular true with reachability properties (because we only perform the forward phase), but also if we expect the property to be false (because we expect the cycle detection phase to terminate early).

The RPG is still interesting with very large state spaces, when we do not have enough memory to store the complete transition relation. We can observe, when we compare the RPG and NO_GRAPH versions, that the time lost re-computing the same transitions several time may not always be too much of a drawback. We see with the PEG model that RPG is substantially better than NO_GRAPH, while the opposite is true for the TK_M model.

A real advantage of the RPG version is to impose no restrictions on the models that are checked. Several model-checking algorithms 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[6]. But this is not always practical, or even possible. This is the case, for example, when model-checking Timed Petri Nets [MF76] using State Class Graphs, a common abstraction for representing a Kripke Structure with real-time constraints; Given a state class, it is only possible to compute a superset of the predecessors. More generally, computing the reverse transition relation is not possible for systems that manipulate data variables. In this case, RPG is the best solution.

To conclude, both RG and RPG can be useful; RPG being the good choice if we are limited by the memory space or we expect the specification to be true. 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.

10 Comparison With Other Tools

We present a comparison of our algorithms with DIVINE [BBCR10], which is the state of the art tool for parallel model checking. More recently, Barnat et al. published that the conjugated use of **[owcty]**

[6] There are also solutions based on recursive traversals of the state graph, but they essentially store the transition relation in the stack, rather than the heap.

and **[map]** (see section 2.1) 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 last distributed version. (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.

In this benchmark, all the configurations (owcty, map, reverse and parental) 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.

Model	Formula	Description	Results
Anderson(an) $18 \cdot 10^6$ states	F1:`(-cs_0) ==> (cs_0)`	If P_0 isn't in CS then it will eventually reach it.	*false*
	F2:`A[]<>(cs_0 or ... or cs_n)`	Infinitely many times someone critical section.	*true*
Lamport (la) $38 \cdot 10^6$ trans.	F1:`(wait_0 and (- cs_0)) ==> (cs_0)`	If P_0 waits for CS then it will eventually get there.	*false*
	F2:`(- cs_0) ==> (cs_0)`	If P_0 isn't in CS then it will eventually reach it.	*false*
	F3:`A[]<>(cs_0 or ... or cs_n)`	Infinitely many times someone in critical section.	*true*
Rether (re) $4 \cdot 10^6$ states	F1:`A[]<>(nrt_0)`	Always some more NRT action of Node 0.	*true*
	F2:`A[]<>(rt_0)`	Always some more RT action of Node 0	*false*
Szymanski (szy) $2 \cdot 10^6$ states	F1:`(wait_0 and (- cs_0)) ==> (cs_0)`	If P_0 waits for CS then it will eventually get there.	*false*
	F2:`(- cs_0) ==> (cs_0)`	If P_0 isn't in CS then it will eventually reach it.	*false*
	F3:`A[]<>(cs_0 or ... or cs_0)`	Infinitely many times someone in critical section.	*true*

Figure 31: Formulas and Models for our Comparison.

Before we continue, it is import to explain the set of selected models. Based on the benchmark presented in [BBCR10], we selected a set of models which there were available valid and non valid formulas. This choice was motivated to establish a broader comparison between our approach, which is "optmized" 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.

Figure 31 depicts the set of models used for this comparison. The first row presents the name, the number of states and transitions of the model. The second row presents the formulas experimented, followed by a short description. The last row depicts the expected result, i.e., if the formula is valid or not.

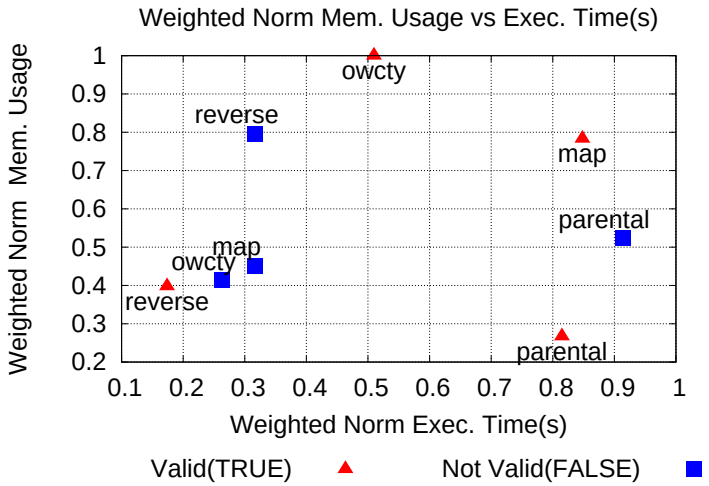


Figure 32: Comparison with divine.

Figure 32 shows the normalized weighted sum of the memory footprint and the execution time obtained in this benchmark. These measures are first normalized and after weighted by the model number of states. 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.

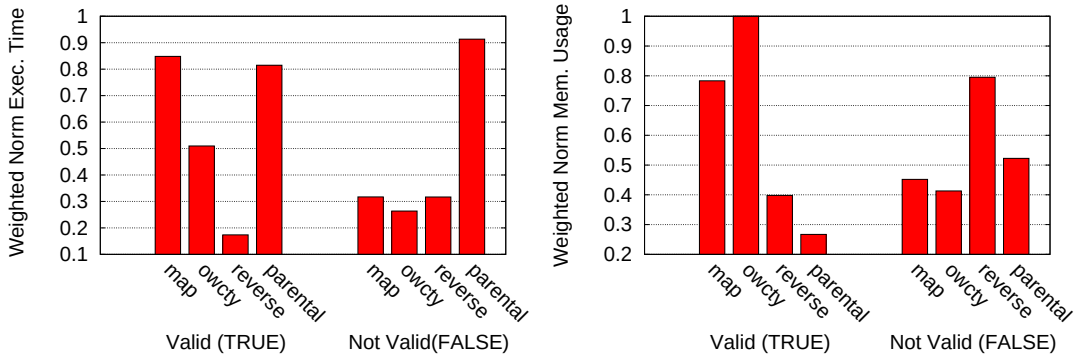


Figure 33: Comparison with divine.

Figure 33 shows the same analysis but using histograms. 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 less memory than map and owcty when the formula is valid. In addition, regardless of its non-linear time complexity, it showed good results compared with map and owcty. This is a good result because parental is able to verify a given formula —in average— using 4 times less memory than owcty by a small amount of extra time ($\approx 60\%$ more).

Now, we present a set of histograms graphs for each individual experiment. We start by presenting the results for the anderson model, followed by the lamport model, the rether model and finally the szymanski model. For each graph we give the execution time (in seconds) and the memory peak.

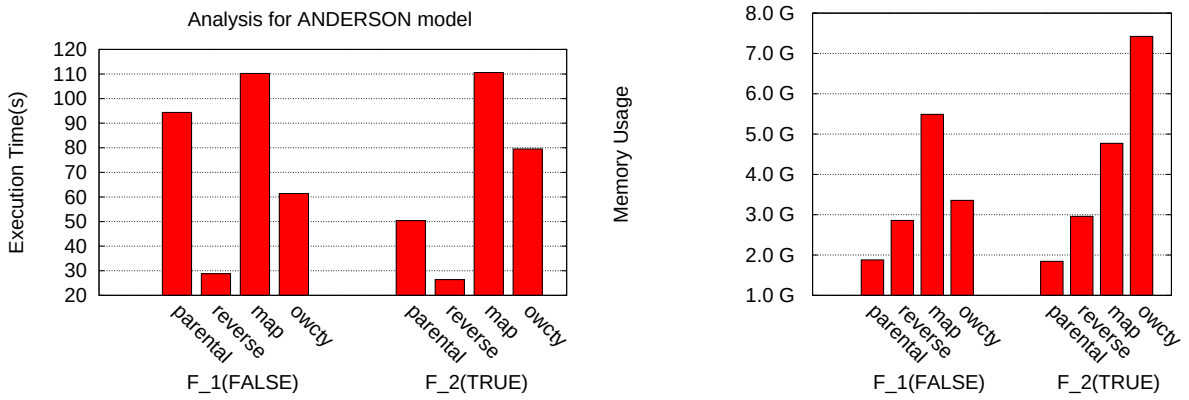


Figure 34: Results for Anderson model.

Figure 34 depicts the results obtained for the anderson model. For this experimented, reverse has the better execution time even when the formula is not valid (see F_1). Regarding the memory used, parental has the most descent usage.

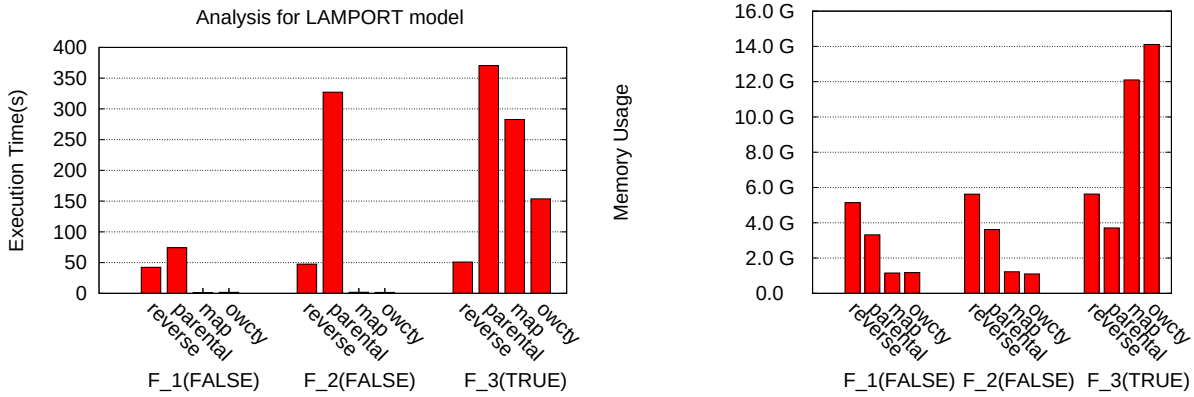


Figure 35: Results for Lamport model.

Figure 35 presents the results for the lamport model. We experimented three formulas: F_1, F_2 and F_3. For the non valid formulas (F_1 and F_2), map and owcty have the best results due to their on-the-fly nature. In these cases, they were able to disproof the formula before the complete construction of the state space. It explains the execution time and memory usage when compared to our algorithms.

However, the on-the-fly profile of map and owcty does not improve the result when the formula is valid. Consequently, we observe for F_3 that reverse holds the best execution time and parental the most descent memory usage.

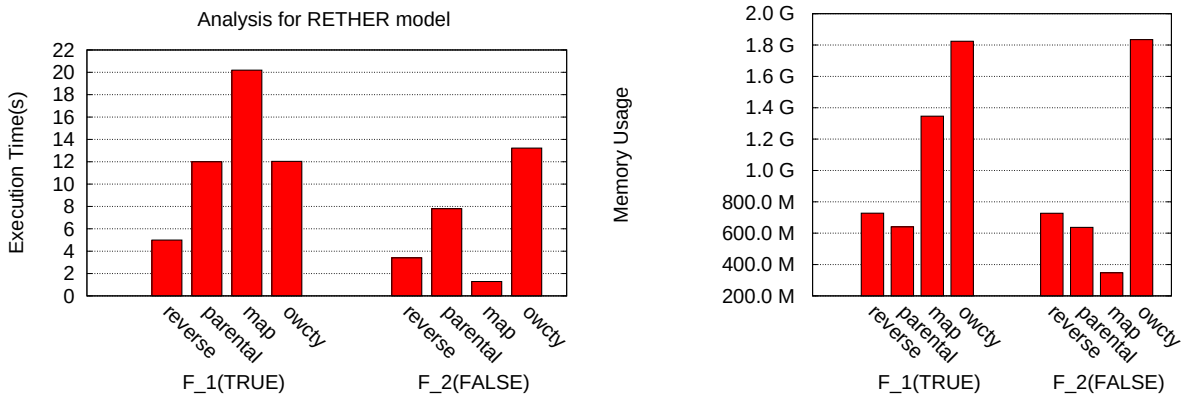


Figure 36: Results for Rether model.

Figure 36 and 37 shows the results for the rether and szymanski models. We observe practically the same results, except for some non valid formulas —F_2 for rether model and F_1 for szymanski— where parental held a better execution time than owcty.

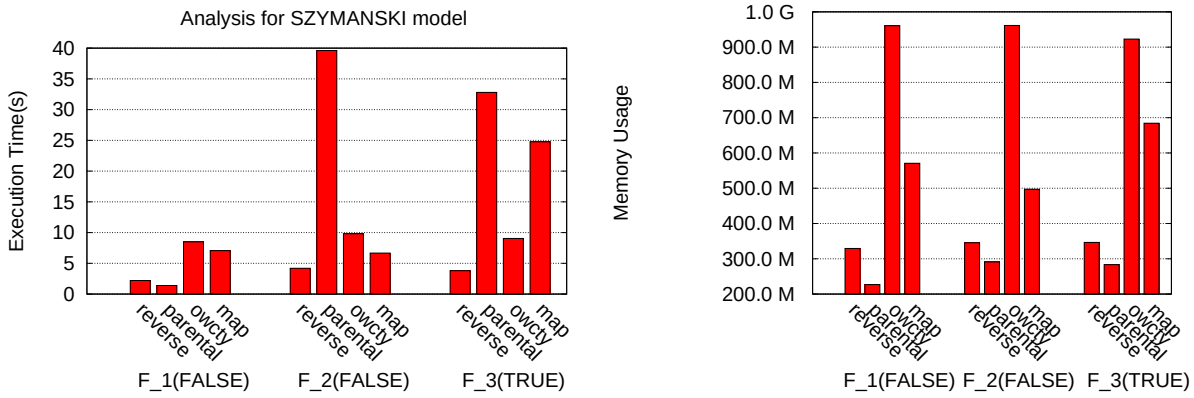


Figure 37: Results for Szymanski model.

To conclude, for the set of models and formulas used in this benchmark, both parental and reverse delivered good results when compared with 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.

These preliminary results against one of the most popular parallel model-checker are very encouraging since we have only a prototype implementation of reverse and parental algorithms.

11 Conclusions

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

We have based our approach on three main principles:

- *Thou shalt not restrict the modelling language*: we only need to be able to compute the successors of a state;
- *Thou shalt not restrict the way states are distributed*: because we should be able to reuse the same algorithm with different state space construction methods; and, finally
- *thou shalt no put restrict the way work is shared among processors*: because the algorithm should play nicely with traditional work-sharing techniques, such as work-stealing or stack-slicing.

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 show that the RG version has a linear time and space complexity ($O(S+R)$), while RPG that has time complexity in $O(S\cdot\cdot(S-R))$ and space complexity in $O(S)$. In these expressions, S stands for the number of reachable states in the system and R for the number of transitions. If we interpret these complexity results using only the number of states—we have R in $O(S^2)$—it is clear that RPG trades computation time (in $O(S^3)$ for RPG against $O(S^2)$ for RG) for memory space (in $O(S)$ for RPG against $O(S^2)$ for RG).

De facto, 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 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. Actually, we already have what is needed to model-check the whole of CTL. Indeed, we can follow the approach proposed by Clarke for CTL model checking [CES86] and reduce the problem of checking a "nested" formula Φ to the problem of checking $|\Phi|$ *basic formulas*. In this context, $|\Phi|$ is an integer value measuring the complexity of the formula Φ—or the number of "sub-formulas" in Φ—and basic formulas correspond to the formulas $E(\phi \cup \psi)$ and $E\Box(\phi)$[7]. A naive implementation of this approach would be to manage $|\Phi|$ copies of our labels (sons and suc) in parallel, but this could have an adverse effect on the memory consumption. At the moment, we are still considering several strategies for model-checking CTL formulas using a bounded number of labels.

[7] we do not consider the modality *next* in this discussion but it is not difficult to add it in our framework.

References

[BBC02] J. Barnat, L. Brim, and I. Cerna. Property driven distribution of nested DFS. In *Proc. Workshop on Verification and Computational Logic*, DSSE Technical Report, page 110, 2002.

[BBC03] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search LTL model-checking. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 106 – 115, October 2003.

[BBCR10] J. Barnat, L. Brim, M. Ceska, and P. Rockai. DiVinE: parallel distributed model checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, page 47. IEEE, 2010.

[BBR07] J. Barnat, L. Brim, and P. Rockai. Scalable multi-core LTL Model-Checking. In *Model Checking Software*, volume 4595 of *LNCS*, page 187203. Springer, 2007.

[BBR09] 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.

[BBS01] Jiri Barnat, Lubos Brim, and Jitka Stribrna. Distributed LTL model-checking in SPIN. In Matthew Dwyer, editor, *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 200–216. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45139-0_13.

[BCKP01] Lubos Brim, Ivana Cerna, Pavel Krcal, and Radek Pelanek. Distributed LTL model checking based on negative cycle detection. In Ramesh Hariharan, V. Vinay, and Madhavan Mukund, editors, *FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science*, volume 2245 of *Lecture Notes in Computer Science*, pages 96–107. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45294-X_9.

[BCMS04] L. Brim, I. Cerna, P. Moravec, and J. Simsa. Accepting predecessors are better than back edges in distributed LTL model-checking. In *Formal Methods in Computer-Aided Design*, page 352366, 2004.

[BCY02] Lubos Brim, Jitka Crhov, and Karen Yorav. Using assumptions to distribute CTL model checking. *Electronic Notes in Theoretical Computer Science*, 68(4):559 – 574, 2002. PDMC 2002, Parallel and Distributed Model Checking (Satellite Workshop of CONCUR 2002).

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, LNCS, page 200236. SpringerVerlag, September 2004.

[BH04] 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.

[BLW01] Benedikt Bollig, Martin Leucker, and Michael Weber. Parallel model checking for the alternation free -Calculus. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 543–558. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45319-9_37.

[BLW02] 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.

[BMBW00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117128, November 2000.

[But97] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[CE82] 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.

[CES86] 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.

[Cla99] E Clarke. *Model checking*. MIT Press, Cambridge Mass., 1999.

[CP03] Ivana Cerna and Radek Pelanek. Distributed explicit fair cycle detection (Set based approach). In Thomas Ball and Sriram Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 623–623. Springer Berlin / Heidelberg, 2003. 10.1007/3-540-44829-2_4.

[CVWY92] 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.

[EJS93] E. Emerson, C. Jutla, and A. Sistla. On model-checking for fragments of -calculus. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-56922-7_32.

[FFK+01] Kathi Fisler, Ranan Fraer, Gila Kamhi, Moshe Vardi, and Zijiang Yang. Is there a best symbolic Cycle-Detection algorithm? In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 420–434. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45319-9_29.

[GHR95] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, USA, 1995.

[HB07] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33:659–674, 2007.

[HJG08a] 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.

[HJG08b] Gerard Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, volume 5156 of *Lecture Notes in Computer Science*, pages 134–143. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-85114-1_11.

[IB02] Cornelia P. Inggs and Howard Barringer. Effective state exploration for model checking on a shared memory architecture. *Electronic Notes in Theoretical Computer Science*, 68(4):605 – 620, 2002. PDMC 2002, Parallel and Distributed Model Checking (Satellite Workshop of CONCUR 2002).

[IB06] 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.

[JM05] Christophe Joubert and Radu Mateescu. Distributed local resolution of boolean equation systems. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, page 264271, Washington, DC, USA, 2005. IEEE Computer Society.

[Kup95] O. Kupferman. *Model Checking for Branching-Time Temporal Logics*. PhD thesis, The Technion, 1995.

[Laf02] A. L Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical 00176, Universität Freiburg, Institute of Computer Science, 2002.

[LLP+11] 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.

[LS99] Flavio Lerda and Riccardo Sisto. Distributed-Memory model checking with SPIN. In Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48234-2_3.

[LvdPW10] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 247 –255, October 2010.

[MC99] Andrew Miner and Gianfranco Ciardo. Efficient reachability set generation and storage using decision diagrams. In Susanna Donatelli and Jetty Kleijn, editors, *Application and Theory of Petri Nets 1999*, volume 1639 of *Lecture Notes in Computer Science*, pages 691–691. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48745-X_2.

[MF76] P. Merlin and D. Farber. Recoverability of communication ProtocolsImplications of a theoretical study. *Communications, IEEE Transactions on*, 24(9):1036 – 1043, September 1976.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 –580, April 1989.

[Pel07] Radek Pelanek. BEEM: benchmarks for explicit model checkers. In *Proceedings of the 14th international SPIN conference on Model checking software*, page 263267, Berlin, Heidelberg, 2007. Springer-Verlag.

[PW08] 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).

[Rei85] John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229 – 234, 1985.

[Saa11] Rodrigo Saad. *Parallel Model Checking for Multiprocessor Architecture*. PhD thesis, L'Institut National des Sciences Appliquées de Toulouse, Toulouse - France, 11/2011 2011.

[Sti99] C. Stirling. Bisimulation, modal logic and model checking games. *Logic Journal of IGPL*, 7(1):103–124, 1999.

[Tar71] Robert Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114 –121, October 1971.

[TSDZB11] 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.

[Wil01] T. Wilke. Alternating tree automata, parity games, and modal m-Calculus. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 8(2):359, 2001.

[Wol86] P. Wolper. An automata-theoretic approach to automatic program verification. In *IEEE Symposium on Logic in Computer Science*, pages 322–331. Computer Society, 1986.