Parallel Model Checking and the FMICS-jETI Platform

Jiří Barnat
Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
barnat@fi.muni.cz

Luboš Brim*
Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
brim@fi.muni.cz

Martin Leucker
Technische Universität München
Boltzmannstr. 3, D-85748 Garching, Germany
leucker@informatik.tu-muenchen.de

Abstract

In this paper we summarize parallel algorithms for enumerative model checking of properties formulated in linear time temporal logic (LTL) as well as a fragment of the μ-calculus which naturally subsumes the branching time logic CTL (computation tree logic). We also indicate how to provide parallel model checking applications as services for integrated modelling, analysis, and verification using the FMICS-jETI platform.

1 Introduction

Conventional model checking techniques have high memory requirements and are very computationally intensive; they are thus unsuitable for handling real-world systems that exhibit complex behaviors which cannot be captured by simple models having a small or regular state space. Various authors have proposed ways of solving this problem by either using powerful shared-memory multiprocessors (e.g. multi-core machines) or by distributing the memory requirements over several machines (e.g. on a cluster of workstations).

The work on parallel verification is quite extensive, growing in recent years. There are attempts to consider both the symbolic as well as the enumerative techniques, theorem-provers as well as sat-solvers, etc. In this paper we focus on enumerative model-checking of temporal properties. More specifically, we summarize model checking of properties formulated in linear time temporal logic (LTL) as well as a fragment of the μ-calculus which naturally subsumes the branching time logic CTL (computation tree logic).

The model-checking problem can always be represented as a problem on a directed graph. We suppose the graph is given implicitly by the function F_{init} returning the initial state and the function F_{succ} returning the set of immediate successors of a given state. This representation allows solving the problem in the "on-the-fly" manner, hence it is often possible to get the answer to the verification problem without actually explicitly generating the entire state space (graph). This is in particular useful in attacking large scale systems.

Model checking traditionally terms the task of verifying an implementation, typically given in terms of a finite-state system, with respect to its specification, typically given as a temporal formula. However, model checking could and probably should also be considered as a flexible analysis tool—as long as the object to analyze is representable as a finite-state system and the analysis can be formulated in a suitable temporal logic. In consequence, model checkers are at the heart of many modelling and analysis tools and will be in the future. It is therefore important to offer easy means for integrating model checkers into other tools.

On a different line, powerful parallel computers are only available at dedicated locations. Nevertheless, parallel model checking applications or the tools built on top of model checkers should be easily usable as conventional desktop applications. Therefore, it is desirable to provide parallel model checking applications as services for direct use and simple integration to customized modelling, analysis, and verification tools.

jETI is a framework that offers such integration capabilities and we discuss how to offer parallel model checking applications with jETI in mind.

*This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/06/1338 and the Academy of Sciences grant No. 1ET408050503.

2 Parallel Reachability Analysis

The basic verification technique is reachability. Reachability is also more amenable for parallelization than the other verification problems and most of the pioneering work in parallel model-checking has been focused on algorithms for verification of safety properties. At the heart of reachability analysis as well as model-checking in general is the state space generation.

Parallel state space generation has been initially studied in the context of Petri nets, stochastic Petri nets, discrete-time and continuous-time Markov chains [32, 19, 38]. Later on, distributed state space exploration algorithms for SPIN [42], Muprhi [53], CADP [30], UPPAAL [6], DIVINE [4], and other tools have been suggested as well. Algorithms for distributed-memory architecture became dominant, primarily due to the easy access of networks of workstations.

All these approaches share a common idea: each machine in the network explores a subset of the state space. The subset is defined using the *partition function*. The function *Partition(s,N)* returns the identifier of the machine to which the state s is assigned, an integer between 0 and N-1. Assuming we have N machines, this function partitions the state space into N classes S^i, one assigned to each machine.

The Algorithm 1([18]) gives the overall idea of the distributed state space generation. The algorithm is supposed to be run on the machine i and the same algorithm is performed on each other machine involved in the distributed computation. The data structure S^i_{new} maintains already generated but not yet processed states. The meaning of other functions and structures is obvious.

The individual algorithms mentioned above differ in a number of design principles and implementation choices such as: the use of internal structures for storing the states (e.g. hash tables or B-trees), the way of partitioning the state space using either static hash functions or dynamic ones that allow dynamic load balancing, etc. Experimental evaluations demonstrate good scalability and speedups obtained are close to linear. Moreover, adaptation to shared-memory architectures does not bring any additional complications and e.g. the SPIN model checker is already expected to provide support for model-checking of safety properties on multi-core machines from its version 4.3.

Algorithm 1 (Distributed State Space Generation)

```
1:  if Partition(Initial,N)=i then
2:      S^i := {Initial}
3:  else
4:      S^i := ∅
5:  end if
6:  S^i_new := S^i
7:  while not received termination do
8:      while ∃s ∈ S^i_new do
9:          S^i_new := S^i_new \ {s}
10:         for each u ∈ F_succs(s) do
11:             j := Partition(u,N)
12:             if i ≠ j then
13:                 SendState(j, u)
14:             else
15:                 if u ∉ S^i then
16:                     S^i_new := S^i_new ∪ {u}
17:                     S^i := S^i ∪ {u}
18:                 end if
19:             end if
20:         end for
21:     end while
22:     S^i_rec := ReceiveStates \ S^i
23:     S^i := S^i ∪ S^i_rec
24:     S^i_new := S^i_new ∪ S^i_rec
25: end while
```

3 Parallel LTL Model Checking

The automata-theoretic approach to model checking finite-state systems against linear-time temporal logic (LTL) uses automata on infinite words to represent both the system and the property to be checked. Both automata are synchronized and the emptiness check for the resulting automaton is performed. The emptiness check problem essentially breaks down to finding reachable accepting cycles in a directed graph G with A as the subset of accepting vertices.

The optimal sequential algorithms for accepting cycle detection use depth-first search strategies to detect accepting cycles. The individual algorithms differ in their space requirements, length of the counter example produced, and other aspects. For a recent survey we refer to [54]. The well-known *Nested DFS* algorithm is used in many model checkers and is considered to be the best suitable algorithm for enumerative sequential LTL model checking. The algorithm was proposed by Courcoubetis et al. [22] and its main idea is to use two interleaved searches to detect reachable accepting cycles. The first search discovers accepting states while the second one, the nested one, checks for self-reachability. Several modifications of the algorithm have been suggested to remedy some of its disadvantages [31]. The time complexity of the algorithm is linear in the size of the graph, i.e. $\mathcal{O}(m+n)$, where m is the number of edges and n is the number of vertices.

The effectiveness of the algorithm *Nested DFS* is achieved due to the particular order in which the graph is explored and which guarantees that vertices are not re-visited more than twice. In fact, all the best-known algorithms rely on the same exploring principle, namely on the *postorder* as

computed by the DFS. It is a well-known fact that the postorder problem is P-complete and, consequently, any parallel algorithm which would be directly based on DFS postorder is unlikely to be efficiently parallelized.

In [34, 35] G. Holzmann and D. Bošnački proposed an adaptation of the Nested DFS algorithm for dual-core machines. The idea is to utilize the independence of the first and the nested search in the Nested DFS algorithm. The algorithm keeps its linear time complexity. On the downside, the algorithm is unable to scale to more than two cores. It is still an open problem to do scalable verification of general liveness properties on N-cores with linear time complexity.

Efficient parallel solution of many problems often requires approaches radically different from those used to solve the same problems sequentially. It seems, that it is extremely difficult to ground parallel LTL model checker on extending the Nested DFS algorithm or any other postorder based algorithm. As we have seen in the previous section, the reachability analysis is a verification problem with efficient parallel solutions. The reason is that the exploration of the state space does not rely on any specific order, the vertices can be visited independently and in any order. The exploration can thus be implemented e.g. using breadth-first search. This gives hope to find good practical solutions for LTL model checking that, though not theoretically optimal, will scale well.

In the following, we overview several other parallel algorithms for enumerative LTL model checking that are all, more or less, based on performing repeated parallel reachability to detect accepting cycles. These algorithms have the potential to scale well, there time complexity is however increased in the general case. The reader is kindly asked to consult the original sources for the details of the presented algorithms.

Algorithm 2 (MAP)

```
1: while A ≠ ∅ do
2:     compute Map    /* max. accepting predecessors */
3:     if (∃u ∈ A : map(u) = u) then
4:         return cycle
5:     else
6:         G := delacc(G)    /* unmark acc. predecessors */
7:     end if
8: end while
9: return no cycle
```

The driving idea of the **Maximal Accepting Predecessor Algorithm (MAP)** [11, 12] is based on the fact that every accepting vertex lying on an accepting cycle is its own predecessor. An algorithm that is directly derived from this idea, would require expensive computation as well as space to store all proper accepting predecessors of all (accepting) vertices. To remedy this obstacle, the MAP algorithm stores only a single representative of all proper accepting predecessor for every vertex.

The representative is chosen as the *maximal accepting predecessor* accordingly to a presupposed linear ordering $\prec$ of vertices (given e.g. by their memory representation). Clearly, if an accepting vertex is its own maximal accepting predecessor, it lies on an accepting cycle. Unfortunately, it can happen that all the maximal accepting predecessor lie outside of accepting cycles.

Such vertexes can be safely deleted from the set of accepting vertexes (by applying the *deleting transformation delacc(G)*) and the accepting cycle still remains in the resulting graph. Whenever the deleting transformation is applied to the graph it shrinks the set of accepting vertices by those vertices that do not lie on any cycle.

As the set of accepting vertices can change after the deleting transformation has been applied, maximal accepting predecessors must be recomputed. It can happen that even in the graph $delacc(G)$ the maximal accepting predecessor function is still not sufficient for cycle detection. However, after a finite number of applications of the deleting transformation an accepting cycle is certified. For a graph without accepting cycles the repetitive application of the deleting transformation results in a graph with an empty set of accepting vertices.

Time complexity of the algorithm is $\mathcal{O}(a^2 \cdot m)$, where a is the number of accepting vertices. Here the factor $a \cdot m$ comes from the computation of the *Map* function and the factor a relates to the number of iterations.

One of the key aspects influencing the overall performance of the algorithm is the underlying ordering of vertices used by the algorithm. In order to optimize the complexity one aims to decrease the number of iterations by choosing an appropriate vertex ordering. Ordering $\prec$ is *optimal* if the presence of an accepting cycle can be decided in one iteration. It can be easily shown that for every (automaton) graph there is an optimal ordering. Moreover, an optimal ordering can be computed in linear time.

An example of an optimal ordering is the depth-first search postorder. Unfortunately, the *optimal ordering problem*, which is to decide for a given graph and two accepting vertices u, v whether u precedes v in *every* optimal ordering of graph vertices, is P-complete [11], hence unlikely to be computed effectively in a distributed environment. Therefore, several heuristics for computing a suitable vertex ordering are used. The trivial one orders vertices lexicographically according to their bit-vector representations. The more sophisticated heuristics relate vertices with respect to the order in which they were traversed. However, experimental evaluation has shown that none of the

heuristics significantly outperforms the others. On average, the most reliable heuristic is the one based on breadth-first search order followed by the one based on (random) hashing.

Algorithm 3 (OWCTY)

1: **while** not finished **do**
2: compute *Reachability* /* remove vertices which are not reachable from accepting vertices */
3: compute *Elimination* /* remove vertices which are not contained in any cycle (have in-degree 0) */
4: **end while**

The inspiration for the next parallel algorithm for detection of accepting cycles is taken from symbolic algorithms for cycle detection, namely from SCC hull algorithms. SCC hull algorithms compute the set of vertices containing all accepting components. The algorithms maintain the approximation of the set and successively remove non-accepting components until they reach a fixpoint. Different strategies to remove non-accepting components lead to different algorithms. An overview, taxonomy, and comparison of symbolic algorithms can be found in independent reports [29] and [49].

The presented algorithm [15] is an adaptation of the **One Way Catch Them Young Algorithm (OWCTY)** [29] to the enumerative setting. The enumerative algorithm works on individual vertices rather than on sets of vertices as is the case in symbolic approach. A component is removed by removing its vertices. The idea of the algorithm is to repeatedly remove vertices from the graph that cannot lie on any accepting cycle. The two removal rules are as follows:

- if a vertex is not reachable from any accepting vertex then the vertex does not belong to any accepting component and
- if a vertex has in-degree zero then the vertex does not belong to any accepting component.

Note that an alternative set of rules can be formulated as

- if no accepting vertex is reachable from a vertex then the vertex does not belong to any accepting component and
- if a vertex has out-degree zero then the vertex does not belong to any accepting component.

This second set of rules results in an algorithm which works in a *backward* manner and we will not describe it explicitly here.

The algorithm performs removal steps as far as there are vertices to be removed. In the end, either there are some vertices left in the graph meaning that the original graph contains an accepting cycle, or all vertices have been removed meaning that the were no accepting cycles in the original graph.

The presented algorithm requires the entire automaton graph to be generated first. Moreover, the backward version actually needs to store the edges to be able to perform backward reachability. This is however payed out by relaxing the necessity to compute successors, which is in fact a very expensive operation in practice.

Time complexity of the algorithm is $\mathcal{O}(h \cdot m)$ where h is the height of the SCC quotient graph. Here the factor m comes from the computation of *Reachability* and *Elimination* functions and the factor h relates to the number of external iterations. In practice, the number of external iterations is very small even for large graphs. This observation is supported by experiments in [29] with the symbolic implementation and hardware circuits problems. Similar results are communicated in [47] where heights of quotient graphs were measured for several models. As reported, 70% of the models has height smaller than 50.

A positive aspect of the algorithm is its effectiveness for *weak automaton graphs*. A graph is weak if each SCC component of G is either fully contained in A or is disjoint with A. For weak graphs one iteration of the algorithm is sufficient to decide existence of accepting cycles. The studies of temporal properties [24, 16] reveal that verification of up to 90% of LTL properties leads to weak automaton graphs.

The algorithm can be effortlessly extended to automaton graphs for other types of nondeterministic word automata like generalized Büchi automata and Streett automata.

Algorithm 4 (BLEDGE)

1: **for each** level = 0 to ... **do**
2: L= all current BL edges
3: **for** $(s,t) \in L$ **do in parallel**
4: test_cycle(s,t,$\mid L \mid$)
5: **end for**
6: **end for**

1: **Proc** *test_cycle*
2: propagate s
3: **if** s propagated to itself **then**
4: **return** cycle
5: **else if** current BL passed $> \mid L \mid$ **then**
6: **return** cycle
7: **end if**

An edge (u, v) is called a *back-level edge* if it does not increase the distance of the target vertex v form the initial vertex of the graph. The key observation connecting the cycle detection problem with the back-level edge concept, as used in the **Back-Level Edges Algorithm (BLEDGE)** [2], is that every cycle contains at least one back-level edge.

Back-level edges are, therefore, used as triggers to start a procedure that checks whether an edge is a part of an accepting cycle. However, this is too expensive to be done completely for every back-level edge. Therefore, several improvements and heuristics have been suggested and integrated within the algorithm to decrease the number of tested edges and speed-up the cycle test.

The BFS procedure which detects back-level edges runs in time $\mathcal{O}(m+n)$. In the worst case, each back-level edge has to be checked to be a part of a cycle, which requires linear time $\mathcal{O}(m+n)$ as well. Since there is at most m back-level edges, the overall time complexity of the algorithm is $\mathcal{O}(m.(m+n))$.

The algorithm performs well on graphs with small number of back-level edges. In such cases the performance of the algorithm approaches the performance of reachability analysis, although, the algorithm performs full LTL model checking. On the other hand, a drawback shows up when a graph contains many back-level edges. In such a case, frequent re-visiting of vertices in the second phase of the algorithm causes the time of the computation to be high.

The level-synchronized BFS approach also allows to involve BFS-based Partial Order Reduction (POR) technique [20] in the computation. POR technique prevents some vertices of the graph from being generated while preserving result of the verification. Therefore, it allows analysis of even larger systems. The standard DFS-based POR technique strongly relies on DFS stack and as such it is inapplicable to cluster-based environment [14].

Algorithm 5 (NEGC)

```
1:  while not finished do
2:      Scan vertices
3:      if successor vertex is accepting then
4:          run walk to root (WTR)
5:          if WTR reaches initial vertex then
6:              continue
7:          else
8:              return cycle
9:          end if
10:     end if
11: end while
```

Consider *maximal number* of accepting vertices on a path from the vertex to a vertex, where the maximum is being taken over all such paths. For vertices on an accepting cycle the maximum does not exist because extending a path along the cycle adds at least one accepting vertex. This opens an idea to detect accepting cycles via maximal numbers of accepting predecessors.

For computing the maximal number of accepting predecessors the algorithm maintains for every vertex v its (current) maximum $d(v)$ giving the maximal number of (so far discovered) accepting predecessors, parent vertex $p(v)$, and status $S(v) \in \{unreached,\ labeled, scanned\}$. Initially, $d(v) = \infty, p(v) = nil$, and $S(v) = unreached$ for every vertex v. The method starts by setting $d(s) = 0, p(s) = nil$ and $S(s) = labeled$, where s is the initial vertex. At every step a *labeled* vertex is selected and scanned. When scanning a vertex u, all its outgoing edges are *relaxed* (immediate successors are checked). Relaxation of an edge (u, v) means that if $d(v)$ is an accepting vertex then $d(v)$ is set to $d(u) + 1$ and $p(v)$ is set to u. The status of u is changed to *scanned* while the status of v is changed to *labeled*. If all vertices are either *scanned* or *unreached* then d gives the maximal number of accepting predecessors. Moreover, the *parent graph* G_p is the graph of these "maximal" paths. More precisely, the parent graph is a subgraph G_p of G induced by edges $(p(v), v)$ for all v such that $p(v) \neq nil$.

Different strategies for selecting a labeled vertex to be scanned lead to different algorithms. When using FIFO strategy to select vertices, the algorithm runs in $\mathcal{O}(m \cdot n)$ time in the worst case. For graphs with reachable accepting cycles there is no "maximal" path to the vertices on an accepting cycle and the scanning method must be modified to recognize such cycles. The algorithm employs the *walk to root* strategy which traverses the *parent graph*. The walk to root strategy is based on the fact (see e.g. [17]) that a cycle in the parent graph G_p corresponds to an accepting cycle in the original graph and vice-versa.

The walk to root method tests whether G_p is acyclic. Suppose the parent graph G_p is acyclic and an edge (u, v) is relaxed, i.e. $d(v)$ is decreased. This operation creates a cycle in G_p if and only if v is an ancestor of u in the current G_p. Before applying the operation, we follow the parent pointers from u until we reach either v or s. If we stop at v a cycle is detected. Otherwise, the relaxation does not create a cycle. However, since the path to the initial vertex can be long, the cost of edge relaxation becomes $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$. In order to optimize the overall computational complexity, amortization is used to pay the cost of checking G_p for cycles. More precisely, the parent graph G_p is tested only after the underlying scanning algorithm performs $\Omega(n)$ relaxations. The running time is thus increased only by a constant factor. The worst case time complexity of the algorithm is thus $\mathcal{O}(n \cdot m)$.

All the algorithms allow for an efficient implementation on a parallel architecture. The implementation is based on partitioning the graph (its vertices) into disjoint parts. Suitable partitioning is important to benefit from parallelization.

One particular technique, that is specific to automata based LTL model checking, is *cycle locality preserving* problem decomposition [3, 41]. The graph (product automaton) originates from synchronous product of the prop-

erty and system automata. Hence, vertices of product automaton graph are ordered pairs. An interesting observation is that every cycle in the product automaton graph emerges from cycles in the system and the property graphs. Let A, B be Büchi automata and $A \otimes B$ their synchronous product. If $\mathcal{C}$ is a strongly connected component in the automaton graph of $A \otimes B$, then A-projection of C and B-projection of C are (not necessarily maximal) strongly connected components in automaton graphs of A and B, respectively.

As the property automaton origins from the LTL formula to be verified, it is typically quite small and can be pre-analyzed. In particular, it is possible to identify all strongly connected components of the property automaton graph. A partition function may then be devised, that respects strongly connected components of the property automaton and therefore preserves cycle locality. The partitioning strategy is to assign all vertices that project to the same strongly connected component of the property automaton graph to the same sub-problem. Since no cycle is split among different sub-problems it is possible to employ localized Nested DFS algorithm to perform local accepting cycle detection simultaneously.

Yet another interesting information can be drawn from the property automaton graph decomposition. Maximal strongly connected components can be classified into three categories:

Type F: (*Fully Accepting*) Any cycle within the component contains at least one accepting vertex. (There is no non-accepting cycle within the component.)

Type P: (*Partially Accepting*) There is at least one accepting cycle and one non-accepting cycle within the component.

Type N: (*Non-Accepting*) There is no accepting cycle within the component.

Realizing that a vertex of the product graph is accepting only if the corresponding vertex in the property automaton graph is accepting it is possible to characterize types of strongly connected components of product automaton graph according to types of components in the property automaton graph. This classification of components into types N, F, and P can be used to gain additional improvements that may be incorporated into the above given algorithms.

All the presented algorithms are implemented in the parallel enumerative LTL model-checker DIVINE [4].

4 Parallel Branching-Time Model-Checking

Famous logics for expressing branching time specifications are both Computation-Tree Logic (CTL, [27]) and Kozen's μ-calculus [39]. The μ-calculus offers boolean combination of formulas and, especially, labelled *next*-state, minimal, and maximal fixpoint quantifiers.

For practical applications, it suffices to restrict the μ-calculus in order to gain tractable model checking procedures. The alternation-free fragment, denoted by L^1_μ, restricts the nesting of minimal and maximal fixpoint operators. Still, it allows the formulation of many *safety* as well as *liveness* properties. While this fragment is already important on its own, it subsumes CTL [27].

Model checking this fragment is linear in the length of the formula as well as the size of the underlying transition system, and several sequential model checking procedures are given in the literature [21, 1, 40, 8]. At the same time, the model checking problem was proven to be P-complete [55, 10], limiting our enthusiasm for finding a (theoretically) good parallel model checking algorithm.

The algorithms can be classified into *global* and *local* algorithms. Global algorithms require that the underlying transition system is completely constructed while local algorithms compute the necessary part of a transition system *on-the-fly*. In plain words, global algorithms typically compute the fixpoints in an inductive manner while the local algorithms decide the problem by a depth-first-search.

Typical on-the-fly model checking algorithms for the μ-calculus [37] are based on a characterization of this problem in terms of two-player games [28, 52]. Then, model checking boils down to establishing the winner when playing on so-called game graphs, which are and-or-graphs enriched with so-called parities. For the alternation-free μ-calculus, these game graphs have a simple structure that allows to determine the winner in parallel efficiently.

A different characterization of the model checking problem can be given in terms of so-called 1-letter-simple-weak-alternating-Büchi automata [40]. However, these are related to games in a straightforward manner [43]. On the same line, one can understand the the model checking problem as solving a boolean equation system [45].

The first parallel model checking algorithm for L^1_μ was presented in [9, 10] and formulated in terms of games. Similar algorithms appeared also in [13] and, reformulated in terms of solving alternating boolean equation systems, in [36]. A slightly different approach for parallel CTL model checking was presented in [7].

The game graph combines states of the transition system and subformulas of the property to check to so-called configurations. Furthermore, plays, which are paths in the game graph, correspond to (tableau-kind) proofs or refutations for the property to check. Plays are either finite or represent an infinite unwinding of a fixpoint formula. Similar as in tableaus, the winner of a finite play is immediate. For example, when reaching a configuration with state s and formula true, the play is one by the protagonist. For infinite plays, an infinite unwinding of minimal fixpoint refutes a

property while an infinite unwinding of a maximal fixpoint proofs the property [52].

The main observation in all parallel algorithms is that the game graph (or the boolean equation system) has a so-called *weak* structure: It can be partitioned into components of a single fixpoint type (either maximal or minimal). These components can be partially ordered and edges of the game graph stay either in the same component or leave the component towards a larger one wrt. the partial order. Thus, every play in this graph gets trapped in a unique component.

The problem of determining whether a play is winning is then divided into two independent problems: One is whether the player wins when entering a component and the second is whether the player can force the play to a specific component.

Thus, one source of parallelism is to determine for each component in bottom-up fashion in parallel the winner for the respective component. This is indicated in Algorithm 6, in which speak of coloring the game graph's configurations into either winning for the protagonist (typically indicated by *green* or winning for the antagonist (indicated by *red*) and use the symbol $\prec$ to denote the order of the components.

Algorithm 6 Main procedure, parallel bottom-up version

```
1: for each component Q_j ∈ 𝒬 in bottom-up order do
2:     for each processor P_i in parallel do
3:         colorizeComponent_i(Q_j)
4:         recolorComponent_i(Q_j)
5:         Propagate colors from initial configurations
           ⌈Q_j⌉ to {Q | Q ≺ Q_j}.
6:     end for
7: end for
```

For the configurations of a single component, the winner can be determined as follows: In each terminal configuration, the winner is immediate. Thus, a simple backwards propagation within the and-or graph in the expected manner gives for most configurations a definite answer. The crucial observation made in [10] is that for all remaining configurations, one player can force to stay on a cycle, on which a fixpoint formula is unwinded. Due to weakness of the game graph, this implies that either a minimal or a maximal fixpoint is unwinded. Thus, all configurations either satisfy the formula or violate the formula at hand. This allows to classify the winner for each configuration in parallel without the need of any communication. Thus, the second source of parallelism is given by distributing each component over the cluster and to first propagate winning information from terminal configuration backwards, in parallel, and then to color all remaining configurations according the component's type.

Algorithm 7 $\mathrm{colorizeComponent}_i(Q_j)$

Colorize those configurations of component owned by processor i.

```
1:  /* start with initial configurations of Q_j */
2:  for each conf ∈ ⌈Q_j⌉ do
3:      processSuccessors(conf, Q_j)
4:  end for
5:  repeat
6:      msg := get(Work_i)
7:      if msg = EXPAND(pred, conf) then
8:          if conf ∉ Conf_i then
9:              processSuccessors(conf, Q_j)
10:             initializeConfiguration(conf)
11:             λ(conf) := color(conf)
12:         end if
13:         if λ_i(conf) ≠ WHITE then
14:             put COLOR(pred, λ_i(conf)), Work_h(pred)
15:         end if
16:         →_i := →_i ∪ {(pred, conf)}
17:     else if msg = COLOR(conf, color) then
18:         decrement count(conf, color)   /* update color
            information */
19:         color' := color(conf)
20:         if color' ≠ λ_i(conf) then
21:             λ_i(conf) := color'
22:             for each pred ∈ pre_i(conf) ∩ Q_j do
23:                 /* only work on current component */
24:                 put COLOR(pred, λ_i(conf)), Work_h(pred)
25:             end for
26:         end if
27:     end if
28: until msg = COMPONENTCOMPLETED
```

In a practical algorithm, the processes of generating the game graph as well as determining the winner for each component are interviewed. The heart of the algorithm is the processing of a single game-graph component Q_j as depicted in Algorithm 7. Given a component (number), it expands all configurations of the component and is called by the main function. As the color information of a terminal node is always immediate, a coloring process is initiated, if a terminal configuration is reached. Colors are then propagated backwards.

The algorithm is designed for a distributed setting. Each processor runs an unmodified copy, and we can only assume a local view of all data structures as explained in Section 2. Thus, we index the local part of a data structure with the number of its "owning" processor (index i for processor P_i).

For processors to communicate among each other, each P_i uses a queue $Work_i$ where processors can deposit requests, for example via some message passing mechanism. The algorithm then continually processes requests from its

queue until the handling of the current component is completed. The locally known configurations of a game graph are stored in set $Conf_i$.

In lines 1–4, the component's initial configurations $\lceil Q_j \rceil$ (the ones with incoming edges from smaller components) are expanded consulting the function F_{succ} (see Section 1). The idea of processSuccessors (line 3, not depicted here) is that if a configuration $conf$ is not yet known, its successors $post(conf)$ are calculated and put on respective work queues. Then the algorithm enters a loop (lines 5–28), where it retrieves the next request msg, and processes it.

In case of a request EXPAND$(pred, conf)$ (lines 7–16) to expand more of the game graph, we check whether the to-be-expanded configuration $conf$ has not yet been seen (line 8). It is then expanded (line 9) and initialized (line 10). A color label $\lambda(conf)$ is determined (line 11). It is then possibly propagated to predecessor $pred$ (lines 13–15). This request is put on the queue of the processor $P_{\mathrm{h}(pred)}$ who is responsible for configuration $pred$. A new game graph edge $(pred, conf)$ is then added (line 16). It is later needed to propagate color changes to predecessor configurations.

We process a coloring request COLOR$(conf, color)$ (lines 17–27) by recording that some successor of configuration $conf$ has just obtained color $color$ (line 18). Then, it is determined whether that color change has impact on $conf$ and its color is updated accordingly (lines 19–21). Also, on color update, the new color is propagated backwards to each predecessor $pre_i(conf) \cap Q_j$ of $conf$ in the current component (lines 22–25).

The processing continues until none of the processors has any requests left to handle, in which the algorithm finishes. This situation is detected by an termination check algorithm (not depicted here) which then inserts a message COMPONENTCOMPLETED into every processor's work queue.

When all processes terminate in line 28, the remaining configurations can be colored in parallel independently by every process (line 4 of the main routine).

While parts of the algorithm sketched above ([10]) are similar to a (sequential) solution of the model checking problem described in [40], it avoids explicit detection of cycles, which is believed hard in parallel. Nevertheless, it meets the optimal linear time bounds of sequential algorithms [10].

The algorithm has been implemented and has been examined by checking live-locks on large industrial examples, which could not be checked before [33].

The algorithm of [10] has been extended in [44] to the richer fragment of the μ-calculus allowing one alternation, denoted by L^2_μ. This fragment is of practical importance since it subsumes LTL [48], as well as CTL* [25], which follows by (unpublished) results from Wolper and [26], and was shown in a direct manner in [23].

The parallel algorithm for L^2_μ employs the algorithm for L^1_μ as a subroutine. Thus, it promises a simple and efficient approach to check formulas of LTL, CTL*, and L^2_μ, though empirical evidence is still future work.

5 Parallel Model Checking and jETI

While traditionally model checking is mainly used for verification of hard- and software systems, it could and probably should also be considered as a flexible analysis tool: The object to analyze is given as a finite-state system and the analysis can be formulated in a suitable temporal logic. Program analysis as model checking [50] or the use of model checking for analyzing biochemical processes [5] are just two examples.

In consequence, model checkers are the heart of many modelling and analysis tools. Furthermore, when designing new applications comprising an analysis that can be formulated as a model checking problem, a cost effective approach will be to integrate a model checker rather than to work out a customized analysis algorithm. It is therefore important to offer the easy integration of model checkers into other tools.

Powerful shared-memory multiprocessor systems and especially powerful clusters of workstations are typically found only at dedicated locations, with skilled administrators maintaining the systems. However, for a user of a model checker, regardless whether she is using the model checker directly or whether she is using a tool built on top of a model checker, it is convenient that the application looks and feels like a typical desktop application: She should not be bothered by running a distributed application, updating to new versions of distributed model checkers, or maintaining a parallel computer. Thus, it is desirable to provide parallel model checking applications as services for direct use and, even more important, integration to customized modelling, analysis, and verification tools.

jETI [51] is a framework that offers such integration capabilities. With jETI, users are able to combine functionalities of tools of different providers, and even from different application domains to solve complex problems that a single tool typically is not able to handle. jETI follows a service-oriented approach that combines heterogeneous services provisioned in different technologies.

Instead of physically integrating tools or libraries in other tools, jETI's integration philosophy is to publish a service that is running remotely at the providers location. Whenever the service is needed, the corresponding provider is consulted. This is ideally for offering distributed model checkers as maintenance of the software as well as of the whole parallel machine is left to the provider of the model checker. Yet, the user of a tool that uses the distributed model checker via jETI may not be aware of using highly

sophisticated and highly maintained systems.

An example for integrating a (sequential) model checker into the jETI framework is given in [46]. Due to jETI's integration philosophy, the integration scheme stays the same even when the model checker is distributed and running remotely on a parallel computer. Thus, using jETI it will be possible to develop high-performance analysis tools based on parallel model checkers, which will also open up a new age for using distributed model checkers.

References

[1] H. R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 11 Apr. 1994.

[2] J. Barnat, L. Brim, and J. Chaloupka. Parallel Breadth-First Search LTL Model-Checking. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 106–115. IEEE Computer Society, 2003.

[3] J. Barnat, L. Brim, and I. Černá. Property Driven Distribution of Nested DFS. In *Proceedinfs of the 3rd International Workshop on Verification and Computational Logic (VCL'02 – held at the PLI 2002 Symposium)*, pages 1–10. University of Southampton, UK, Technical Report DSSE-TR-2002-5 in DSSE, 2002.

[4] J. Barnat, L. Brim, I. Černá, P. Moravec, P. Ročkai, and P. Šimeček. DiVinE – A Tool for Distributed Verification (Tool Paper). In *Computer Aided Verification*, volume 4144 of *LNCS*, pages 278–281. Springer, 2006.

[5] G. Batt, D. Bergamini, H. de Jong, H. Garavel, and R. Mateescu. Model checking genetic regulatory networks using gna and cadp. In S. Graf and L. Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 158–163. Springer, 2004.

[6] G. Behrmann, T. S. Hune, and F. W. Vaandrager. Distributed timed model checking — how the search order matters. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 216–231. Springer-Verlag, 2000.

[7] A. Bell and B. R. Haverkort. Sequential and distributed model checking of petri net specifications. In *Proceedings of the 1st Workshop on Parallel and Distributed Methods for Verification*, volume 68 of *Electronic Notes in Theoretical Computer Science*, 2002.

[8] G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings, 11^{th} Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[9] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation free μ–calculus. Technical Report AIB-04-2001, RWTH Aachen, Mar. 2001.

[10] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation-free mu-calculus. In *Proceedings of the 9th International SPIN Workshop on Model checking of Software (SPIN '02)*, volume 2318 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., 2002.

[11] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.

[12] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC'05)*, July 2005.

[13] L. Brim, J. Crhova, and K. Yorav. Using assumptions to distribute CTL model checking. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[14] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to order vertices for distributed ltl model-checking based on accepting predecessors. volume 135.2 of *Electronic Notes in Theoretical Computer Science*, pages 3–18. Elsevier, 2006.

[15] I. Černá and R. Pelánek. Distributed Explicit Fair cycle Detection (Set Based Approach). In *Model Checking Software. 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.

[16] I. Černá and R. Pelánek. Relating Hierarchy of Temporal Properties to Model Checking. In *Proc. Mathematical Foundations of Computer Science*, volume 2747 of *LNCS*, pages 318–327. Springer, 2003.

[17] B. V. Cherkassky and A. V. Goldberg. Negative-Cycle Detection Algorithms. *Mathematical Programming*, 85:277–311, 1999.

[18] G. Ciardo, J. Gluckman, and D. Nicol. Distributed State-Space Generation Of Discrete-State Stochastic Models. Technical Report TR-95-75, Institute for Computer Applications in Science and Engineering, 1995.

[19] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. *INFORMS J. Comp.*, 10(1):82–93, 1998.

[20] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[21] R. Cleaveland and B. Steffen. A linear–time model–checking algorithm for the alternation–free modal mu–calculus. In K. G. Larsen and A. Skou, editors, *Proceedings of Computer-Aided Verification (CAV'91)*, volume 575 of *Lecture Notes in Computer Science*, pages 48–58, Berlin, Germany, July 1992. Springer.

[22] C. Courcoubetics, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification: Proc. of the 2nd International Conference CAV'90*, pages 233–242. Springer, 1991.

[23] M. Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, Apr. 1994.

[24] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-State Verification. In *Proc. Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, 1998.

[25] E. Emerson and J. Y. Halpern. 'Sometimes' and 'Not Never' revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1985.

[26] E. Emerson and C. Lei. Efficient model checking in fragments of the propositional μ–calculus. In *Symposion on Logic in Computer Science*, pages 267–278, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[27] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, Dec. 1982.

[28] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *Proc. 5th International Computer-Aided Verification Conference*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 1993.

[29] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 420–434. Springer, 2001.

[30] H. Garavel, R. Mateescu, and I. Smarandache. Parallel State Space Construction for Model-Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.

[31] J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *TACAS'04*, volume 2988 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2004.

[32] B. R. Haverkort, A. Bell, and H. C. Bohnenkamp. On the efficient sequential and distributed generation of very large Markov chains from stochastic Petri nets.

[33] F. Holmén, M. Leucker, and M. Lindström. UppDMC – a distributed model checker for fragments of the μ-calculus. volume 128/3 of *Electronic Notes in Computer Science*. Elsevier Science Publishers, 2004.

[34] G. Holzmann. The Design of a Distributed Model Checking Algorithm for SPIN. In *FMCAD, Invited Talk*, 2006.

[35] G. Holzmann and D. Bosnacki. Multi-core model checking with Spin. In *HIPS-TopModels 2007, short paper*, 2007.

[36] C. Joubert and R. Mateescu. Distributed local resolution of boolean equation systems. In *PDP*, pages 264–271. IEEE Computer Society, 2005.

[37] M. Jurdzinski. Small progress for solving parity games. In *Proc. STACS*, volume 1770 of *LNCS*, pages 290–301. Springer-Verlag, 2000.

[38] W. Knottenbelt, P. Harrison, M. Mestern, and P. Kritzinger. A probabilistic dynamic technique for the distributed generation of very large state spaces. *Performance Evaluation Journal*, 39(1–4):127–148, February 2000.

[39] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, Dec. 1983.

[40] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, Mar. 2000.

[41] A. L. Lafuente. Simplified distributed LTL model checking by localizing cycles. Technical Report 00176, Institut für Informatik, University Freiburg, Germany, July 2002.

[42] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software (SPIN'99)*, volume 1680 of *LNCS*, pages 22–39. Springer, 1999.

[43] M. Leucker. Model checking games for the alternation free mu-calculus and alternating automata. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning "(LPAR'99)"*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1999.

[44] M. Leucker, R. Somla, and M. Weber. Parallel model checking for LTL, CTL* and L^2_μ. In L. Brim and O. Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier Science Publishers, 2003.

[45] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technische Universität München, 1996.

[46] T. Margaria, R. Nagel, and B. Steffen. Remote integration and coordination of verification tools in jeti. In *Proc. of the 12th IEEE Int. Conf. on the Engineering of Computer-Based Systems (ECBS 2005)*, pages 431–436. IEEE Computer Society, 2005.

[47] R. Pelánek. Typical structural properties of state spaces. In *Proc. of SPIN Workshop*, volume 2989 of *LNCS*, pages 5–22. Springer, 2004.

[48] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, Oct. 31–Nov. 2 1977. IEEE Computer Society Press.

[49] K. Ravi, R. Bloem, and F. Somenzi. A Comparative Study of Symbolic Algorithms for the Computation of Fair Cycles. In *Proc. Formal Methods in Computer-Aided Design*, volume 1954 of *LNCS*, pages 143–160. Springer, 2000.

[50] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In G. Levi, editor, *Proc. 5th Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–381. Springer, 1998.

[51] B. Steffen, T. Margaria, and R. Nagel. jeti: A tool for remote tool integration. In N. Halbwachs and L. D. Zuck, editors, *Proc. of 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *LNCS*, Edinburgh, UK, April 2005. Springer Verlag.

[52] C. Stirling. Games for bisimulation and model checking, July 1996. Notes for Mathfit Workshop on finite model theory, University of Wales, Swansea,.

[53] U.Stern and D. L. Dill. Parallelizing the murφ verifier. In O. Grumberg, editor, *Proceedings of Computer Aided Verification (CAV '97)*, volume 1254 of *LNCS*, pages 256–267. Springer, 1997.

[54] M. Vardi. Automata-Theoretic Model Checking Revisited. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2007)*, volume 4349 of *LNCS*. Springer, 2007.

[55] S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science*, pages 154–163, Paris, France, 4–7 July 1994. IEEE Computer Society Press.