Local Parallel Model Checking for the Alternation-Free μ-Calculus

Benedikt Bollig[1], Martin Leucker[2], and Michael Weber[1]

[1] Lehrstuhl für Informatik II, RWTH Aachen, Germany
{bollig, michaelw}@informatik.rwth-aachen.de
[2] Computer and Information Science, University of Pennsylvania, USA
leucker@cis.upenn.edu

Abstract. We describe the design of (several variants of) a local parallel model-checking algorithm for the alternation-free fragment of the μ-calculus. It exploits a characterisation of the problem for this fragment in terms of two-player games. For the corresponding winner, our algorithm determines in parallel a winning strategy, which may be employed for interactively debugging the underlying system, and is designed to run on a network of workstations. Depending on the variant, its complexity is linear or quadratic. A prototype implementation within the verification tool TRUTH shows promising results in practice.

1 Introduction

Model checking [8] is a key tool for the verification of complex hardware and software systems. However, the so-called *state-space explosion* still limits its application. While *partial-order reduction* or *symbolic model checking* reduce the state space by orders of magnitude, typical verification tasks still take modern sequential computers to their memory limits. On the other hand, cheap yet powerful parallel computers can be constructed of Networks Of Workstations (*NOW*s). From the outside, a NOW appears as one single parallel computer with high computing power and, even more important, huge amount of memory. This enables parallel programs to utilise the accumulated resources of a NOW to solve large problems. Hence, it is important to find parallel model-checking algorithms, which then may be combined with well-known techniques to avoid the state-space explosion gaining even more speedup and further reduce memory requirements.

A well-known logic for expressing specifications is Kozen's μ-calculus [14], a temporal logic offering Boolean combination of formulae and, especially, labelled *next*-state, minimal fixed-point, and maximal fixed-point quantifiers. For practical applications, however, it suffices to restrict the μ-calculus to the alternation-free fragment, denoted by L_μ^1, in which nesting of minimal and maximal fixed-point operators is prohibited. It allows the formulation of many *safety* as well as *liveness* properties. It subsumes the logic CTL [10] which is employed in many practical verification tools. It can be shown that the model-checking problem for this fragment is linear in the length of the formula as well as the size of the underlying transition system. Several sequential model-checking procedures are given in the literature (cf. [4] for an overview). The algorithms can be classified into *global* and *local* algorithms. The first require the underlying transition system to be completely constructed while local ones compute the necessary part of a transition system *on-the-fly*.

In complexity theory, it is a well-accepted view that P-complete problems are so-called *inherently sequential*. It was shown in [23, 15, 5] that model checking L^1_μ is P-complete. Thus, all we can hope is to find a linear-time algorithm and no one in the parallel complexity class NC, unless NC equals P. We present a parallel local model-checking algorithm and several of its variations which have linear or quadratic time complexity, thus matching the perfect bounds. We implemented the algorithm within our verification tool TRUTH [17] and learned that it behaves well for practical problems.

Our algorithm uses a characterisation of the model-checking problem for this fragment in terms of two-player games [11, 21]. Strictly speaking, we present a parallel algorithm for colouring a so-called game graph answering the underlying model-checking problem. We show that the game graph has a certain characteristic structure when considering the alternation-free μ-calculus. This is one of the crucial observations and guides us to define a sequential algorithm (without cycle detection) that can easily be parallelised, which we do to obtain our parallel model-checking algorithm. Furthermore, we explain how to extend our algorithm for computing winning strategies without further costs. A strategy may be employed by the user of a verification tool for debugging the underlying system interactively [21].

A different characterisation of the model-checking problem can be given in terms of so-called 1-letter-simple-weak alternating Büchi automata [15]. However, these are related to games in a straightforward manner [11, 16]. Hence, our algorithm can also be understood as a parallel procedure for checking the emptiness of these automata, thus, also as an automata-theoretic model-checking algorithm.

Indeed, our parallel algorithm is inspired by a solution of the model-checking problem described in [15]. However, the proposed algorithm employs a detection of cycles, which is unlikely to be parallelised in a simple way. Our key observation is that we can omit this step yielding a simple parallel algorithm. Note that the game graph is also a Boolean graph and that our algorithm has similarities with the ones of [1, 18].

Until today, not much effort has been taken to consider parallel model-checking algorithms. In [20], a parallel reachability analysis is carried out. The distribution of the underlying structure is similar to the one presented here. But their algorithm is not suitable for model checking temporal logic formulae. [13, 22, 2] present parallelised data structures which employ further computers within a network as a substitute for external storage. The algorithms described in [19, 7] divide the underlying problem into several tasks. However, they are designed in the way that only a single computer can be employed to sequentially handle one task at a time. In [5], we presented a parallel algorithm for a fragment of L^1_μ. [12] introduced a symbolic parallel algorithm for the full μ-calculus. However, both are global so that the transition system has to be constructed totally. [6] presents a model-checking algorithm for LTL using a costly parallel cycle detection. Confer [4] for further related work. Our main contribution is the first *local* parallel model-checking algorithm for L_μ^1 that supports interactive debugging and omits a cycle detection, which allows a powerful parallel realisation of it.

In Section 2, we fix some notions on graphs, recall the syntax and semantics of the μ-calculus as well as the definition of model checking. Furthermore, we describe model-checking games for the μ-calculus and provide an important characterisation of the game graph which will be the basis for our our sequential and parallel algorithms. To simplify our presentation, we start in Section 3 with the presentation of sequential model-checking algorithms that admit a simple parallel version. The corresponding parallel model-checking procedure is shown in Section 4. Before we draw the conclusion of our approach, we present our experimental results in Section 5. A full version of the paper including precise definitions, proofs, and further explanations is available in [4].

2 Graphs, μ-Calculus, and Games

Graphs A tree order is a pair $(Q, \leq)$ such that $\leq$ is a partial ordering relation on Q and its covering relation is a tree. More specifically, assume $\leq$ is a reflexive, antisymmetric, and transitive relation and $\lessdot = \leq - (\leq \circ \leq)$ its *covering relation*. We call $\leq$ a *tree order* iff $\lessdot$ is a tree in the usual sense. Notions of *parents* and *children* for elements of Q wrt. $\leq$ correspond to the usual ones for elements of Q wrt. $\lessdot$.

A directed *graph* $\mathcal{G}$ is a tuple $\mathcal{G} = (Q, \rightarrow)$ where Q is a set of *nodes* and $\rightarrow \subseteq Q \times Q$ is the set of (directed) *edges*. We use notions as *path*, *cycle*, (strongly connected) components, *(induced) subgraphs* as usual. Let $\mathcal{G}' = (Q', \rightarrow')$ and $\mathcal{G}'' = (Q'', \rightarrow'')$ be two components of $\mathcal{G}$ with $Q' \cap Q'' = \emptyset$. Assume that $\rightarrow \cap (Q'' \times Q') = \emptyset$. Then every edge from a node $q' \in Q'$ to a node $q'' \in Q''$ $(q' \rightarrow q'')$ is called a *bridge*.

In the next sections, we consider graphs that are labelled by formulae. We say that a cycle *contains* a formula φ iff the cycle contains a node labelled by φ.

$Q_1, \dots, Q_m$ is a *tree decomposition* of a graph $(Q, \rightarrow)$ iff the Q_i form a partition of Q, i.e., $Q = \bigcup_{i \in \{1,\dots,m\}} Q_i$ and for all $i, j \in \{1, \dots, m\}$ with $i \neq j$, it holds $Q_i \cap Q_j = \emptyset$, and furthermore, there exists a tree order $\leq$ on the collection of the Q_i's such that we have $Q_i \lessdot Q_j$ iff there is a bridge from Q_i to Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$.

The μ-Calculus Let *Var* be a set of fixed-point variables and $\mathcal{A}$ a set of actions. Formulae of the modal μ-calculus over *Var* and $\mathcal{A}$ in positive form as introduced by [14] are defined as follows:

$$\varphi ::= \texttt{false} \mid \texttt{true} \mid X \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [K]\varphi \mid \langle K \rangle \varphi \mid \nu X.\varphi \mid \mu X.\varphi$$

where $X \in \mathit{Var}$ and $K \subseteq \mathcal{A}$.[1] For a formula φ of the μ-calculus, we introduce the notion of *subformulae* (denoted by $\mathit{Sub}(\varphi)$), *free* and *bound* variables, and

[1] $\langle - \rangle \varphi$ is an abbreviation for $\langle \mathcal{A} \rangle \varphi$.

sentences as usual. We call φ a μ-formula iff $\varphi = \mu X.\psi$ for appropriate X and ψ. ν-formulae are introduced analogously. From now on, we assume all formulae to be sentences.

A formula φ is *normal* iff every occurrence of a binder μX or νX in φ binds a distinct variable. For example, $(\mu X.X) \vee (\mu X.X)$ is not normal but $(\mu X.X) \vee (\mu Y.Y)$ is. By renaming, every formula can easily be converted into an equivalent normal formula. If a formula φ is normal, every (bound) variable X of φ *identifies* a unique subformula $\mu X.\psi$ or $\nu X.\psi$ of φ where X is a free variable of ψ. We call X a ν-variable iff it identifies a ν-formula, and μ-variable otherwise. From now on, we assume all formulae to be normal.

For rest of the paper, let us fix a *labelled transition system* $\mathcal{T} = (S, T, \mathcal{A}, s_0)$ where S is a finite set of states, $\mathcal{A}$ a set of actions, and $T \subseteq S \times \mathcal{A} \times S$ denotes the transitions. As usual, we write $s \overset{a}{\to} t$ instead of $(s, a, t) \in T$. Furthermore, let $s_0 \in S$ be the *initial state* of the transition system. The satisfaction of φ wrt. $\mathcal{T}$ and a state $s \in S$ is denoted by $\mathcal{T}, s \models \varphi$ and defined as usual [21, 4].

We use identifiers like $\varphi, \psi, \ldots$ for formulae, $s, t, \ldots$ for states, and $a, b, \ldots$ for actions of the transition system under consideration. K denotes a set of actions. Whenever the sort of the fixed point does not matter, we use σ for either μ or ν.

Essential for our further development is a formula's graph representation. To simplify the definition, let us recall its tree (term) representation. Let φ be a formula. The *occurrence set* of φ, denoted by $Occ(\varphi)$, is inductively defined by $\epsilon \in Occ(\varphi)$, $i\pi \in Occ(\varphi)$ if $i \in \{1, 2\}$, $\varphi = \varphi_1 \star \varphi_2$, and $\pi \in Occ(\varphi_i)$, and $1\pi \in Occ(\varphi)$ if $\varphi = \#\varphi_1$ and $\pi \in Occ(\varphi_1)$, where $\star$ denotes a binary and $\#$ a unary operator. Let $\varphi|_\pi$ denote the subformula of φ at position π, that is $\varphi|_\epsilon = \varphi$ and $\varphi|_{i\pi} = \varphi_i|_\pi$ where $i \in \{1, 2\}$ and $\varphi = \varphi_1 \star \varphi_2$, or $i = 1$ and $\varphi = \#\varphi_1$.

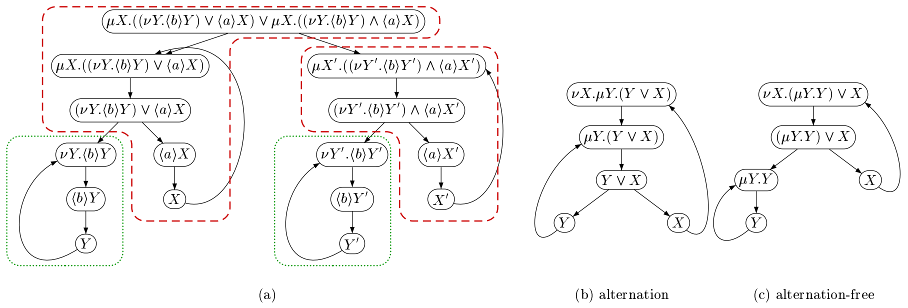
We can now assign to every φ a $Sub(\varphi)$-labelled tree with nodes $Occ(\varphi)$ and edge set $\to$ defined by $\to = \{(\pi, i\pi) \mid \pi, i\pi \in Occ(\varphi), \pi \in \mathbb{N}^*, i \in \mathbb{N}\}$. The labels are assigned in the expected manner by $\lambda(\pi) = \varphi|_\pi$. Altogether, $\mathcal{T}(\varphi) = (Occ(\varphi), \to, \lambda)$ is defined to be the *tree representation* of φ.

We are now ready to define the graph representation of a formula φ. Basically, a formula's graph is its canonical tree representation enriched by edges from fixed-point variables back to the fixed-point formula it identifies.

Definition 1. *Let φ be a formula of the μ-calculus and $\mathcal{T}(\varphi) = (Occ(\varphi), \to, \lambda)$ its tree representation. The* graph of φ*, denoted by $\mathcal{G}(\varphi)$, is $(Occ(\varphi), \to', \lambda)$ where $\to' = \to \cup \{(\pi, \pi') \mid \lambda(\pi) = X$ and $\lambda(\pi') = \sigma X.\varphi'$ for $X \in Var$ and appropriate $\varphi'\}$.*

The graph of the formula $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$ is shown in Figure 1(a).

The *alternation-free fragment* of the μ-calculus is the sublogic of the μ-calculus where no subformula ψ of a formula φ contains both a free variable X bound by a μX in φ as well as a free variable Y bound by a νY in φ. In terms of the graph representation, a formula φ is alternation free iff $\mathcal{G}(\varphi)$ contains no cycle with a ν-variable as well as a μ-variable. Figure 1(b) shows the graph of an



$\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X.((\nu Y.\langle b\rangle Y) \wedge \langle a\rangle X)$
$\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X)$
$(\nu Y.\langle b\rangle Y) \vee \langle a\rangle X$
$\nu Y.\langle b\rangle Y$
$\langle b\rangle Y$
Y
$\langle a\rangle X$
X
$\mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$
$(\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X'$
$\nu Y'.\langle b\rangle Y'$
$\langle b\rangle Y'$
Y'
$\langle a\rangle X'$
X'
(a)
$\nu X.\mu Y.(Y \vee X)$
$\mu Y.(Y \vee X)$
$Y \vee X$
Y
X
(b) alternation
$\nu X.(\mu Y.Y) \vee X$
$(\mu Y.Y) \vee X$
$\mu Y.Y$
Y
X
(c) alternation-free

Fig. 1. Graphs of formulae

alternating formula which has a cycle containing X as well as Y. Contrary, Figure 1(c) shows the graph of an alternation-free formula which has two maximal strongly connected components, one on which X occurs, a second containing Y.

An essential observation is that the graph of an alternation-free formula can naturally be decomposed into so-called μ- and ν-components (cf. Figure 1(a)).

Theorem 1. *Let φ be an alternation-free formula that contains at least one fixed-point formula and let $\mathcal{G}(\varphi) = (Q, \rightarrow, \lambda)$ be its graph representation. Then there exists a tree decomposition $Q_1, \ldots, Q_m$ of $\mathcal{G}(\varphi)$ such that every subgraph induced by Q_i either contains only μ-cycles (called μ-component in the following) or only ν-cycles (ν-component).*

In our graphical representation, the previous components are enclosed by a (red) dashed line and the latter by a (green) dotted line. Note that the components are no-longer necessarily strongly connected. For formulae without any fixed-point formula, we will get a single component, which we call arbitrarily a μ-component.

Proof. Consider the nodes of maximal non-trivial strongly connected components. Alternation freeness guarantees that not both a ν-variable as well as a μ-variable is reached on a cycle in such a component. It is now easy to see that all remaining nodes form trees. Since a formula's graph is connected, the strongly connected components can be canonically ordered by bridges. Please note that maximality of the strongly connected components and trees guarantees the order defined to be a tree order. To obtain the required type of components, strongly connected components are united with their children (components) that are trees. If the root component is a tree, this is united with the first strongly connected component. See [4] for details.

Note that, for the previous decomposition, it is essential that we distinguish between syntactically identical subformulae, which is achieved by using occurrences of formulae instead of directly formulae. In Section 4, we will discuss an alternative definition of a formula's graph which yields partially ordered but no longer tree ordered components and, as we will see, a slightly different parallel algorithm.

It is easy to see that the time complexity of computing the decomposition is linear wrt. the formula's length. Thus, we can label every subformula of a formula with its component number within linear time wrt. the length of the formula.

Model-checking games for the μ-calculus Let us recall Stirling's characterisation of the model-checking problem in terms of games [21]. As we will see, deciding whether a given transition system satisfies a formula is reduced to the colouring of a structure called *game graph*. We will explain that the decomposition of a formula's graph induces a decomposition of the game graph. The latter will simplify our sequential as well as parallel colouring algorithm. The experienced reader will notice that our definition is a little different than Stirling's original approach. We do so to obtain a tree-like decomposition of the game graph instead of a more general dag-like decomposition (cf. Section 4).

Consider the transition system $\mathcal{T}$ and the formula φ. The *model-checking game* of $\mathcal{T}$ and φ has as the board the Cartesian product $S \times Q$ of the set of states and φ's positions. The game is played by two players, namely $\forall$belard (the pessimist), who wants to show that $\mathcal{T}, s_0 \models \varphi$ does *not* hold, whereas $\exists$loise (the optimist) wants to show the opposite.

The model-checking game $G(s, \varphi)$ for a state s and a formula φ is given by all its *plays*, i.e. (possibly infinite) sequences $C_0 \Rightarrow_{P_0} C_1 \Rightarrow_{P_1} C_2 \Rightarrow_{P_2} \ldots$ of *configurations*, where for all i, $C_i \in S \times Q$, $C_0 = (s, \epsilon)$, and P_i is either $\exists$loise or $\forall$belard. We write $\Rightarrow$ instead of $\Rightarrow_{P_i}$ if we abstract from the players. Each next turn is determined by the current subformula of φ. Hence, the label of the second component of a configuration C_i determines the player P_i who has to choose the next move. $\forall$belard makes universal $\Rightarrow_\forall$-moves, $\exists$loise makes existential $\Rightarrow_\exists$-moves. More precisely, whenever $C_i = (s, \pi)$ and

1. $\lambda(\pi) = \texttt{false}$, then the play is finished.
2. $\lambda(\pi) = \psi_1 \wedge \psi_2$, then $\forall$belard chooses $j = 1$ or $j = 2$, and $C_{i+1} = (s, \pi j)$.
3. $\lambda(\pi) = [K]\psi$, then $\forall$belard chooses a transition $s \overset{a}{\to} t$ with $a \in K$ and $C_{i+1} = (t, \pi 1)$.
4. $\lambda(\pi) = \nu X.\psi$, then $C_{i+1} = (s, \pi 1)$.

If $\lambda(\pi) \in \{\texttt{true}, \psi_1 \vee \psi_2, \langle K \rangle \psi\}$ (moves 5–8), it is $\exists$loise's turn; her rules are dually defined to the ones for $\forall$belard. For $\lambda(\pi) = X$, let π' be the position of the μ-/ν-formula X identifies, then $C_{i+1} = (s, \pi')$ (move 9). We will speak of $\forall$*belard-moves* in cases 1–4 and 9, and $\exists$*loise-moves* in all other cases. C_i is called $\forall$-configuration or $\exists$-configuration, respectively (cf. [4]).

A configuration is called *terminal* if no (further) move is possible. A play G is called *maximal* iff it is infinite or ends in a terminal configuration. The *winner* of a maximal play is defined in the following way: If the play is finite, thus ending in a configuration (s, π), then $\forall$belard wins G iff $\lambda(\pi) = \texttt{false}$ or $\lambda(\pi) = \langle K \rangle \psi$.[2] Dually, $\exists$loise wins G iff $\lambda(\pi) = \texttt{true}$ or $\lambda(\pi) = [K]\psi$.[2] An

[2] Note that due to maximality we have $\nexists t \; : \; s \overset{a}{\to} t$ for any $a \in K$.

infinite play is won by ∀belard iff the outermost fixed point that is unwinded infinitely often is a μ-fixed point. Otherwise, when the outermost fixed point that is unwinded infinitely often is a ν-fixed point, then ∃loise wins the game.

A *strategy* is a set of rules for a player P telling her or him how to move in the current configuration. It is called *history free*, if the strategy only depends on the current configuration without considering the previous moves. A *winning strategy* guarantees that the play that P plays according to the rules will be won by P. [21] shows that model checking for the μ-calculus is equivalent to finding a history-free winning strategy for one of the players: Let $\mathcal{T}$ be a transition system with state s and φ a μ-calculus formula. $\mathcal{T}, s \models \varphi$ implies that ∃loise has a history-free winning strategy starting in (s, φ), and $\mathcal{T}, s \not\models \varphi$ implies that ∀belard has a history-free winning strategy starting in (s, φ).

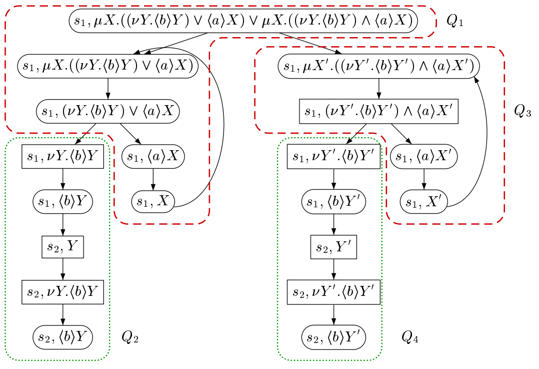
All possible plays for a transition system $\mathcal{T}$ and a formula φ are captured in the *game graph* whose nodes are the elements of the game board (the possible configurations) and whose edges are the players' possible moves. The game graph can be understood as an *and-/or*-graph where the *or*-nodes (denoted by $\bigvee$) are ∃-configurations and the *and*-nodes (denoted by $\bigwedge$) are ∀-configurations.

The following characterisation of the game graph for this fragment is essential for formulating our sequential and parallel algorithms, but only holds for the alternation-free μ-calculus.

Theorem 2. *Let $\mathcal{T}$ be a labelled transition system and let φ be a formula of the alternation-free μ-calculus. Furthermore, let $\mathcal{G} = (Q, E)$ be their game graph. Then there exists a tree decomposition $Q_1, \ldots, Q_m$ of $\mathcal{G}$ such that in every subgraph induced by Q_i, either μ-formulae and no ν-formulae are unwinded or ν-formulae and no μ-formulae. We call Q_i μ-component or ν-component, resp.*

Proof. By Theorem 1, φ's graph admits a decomposition into either μ- or ν-components $Q'_1, \ldots, Q'_m$. Let Q_i be the set of configurations whose formulae are in Q'_i. It is a simple task to show that the Q_i have the desired properties.

Figure 2 shows a game graph for a transition system that has two states s_1 and s_2, an a-loop from s_1 to itself, and a b-edge from s_1 to s_2, and the formula $\Phi = \mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$. ∀belard-configurations are marked by rectangular boxes while ∃loise-configurations are drawn as oval nodes. The dashed and dotted lines identify μ-components and respectively ν-components.



$s_1, \mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X.((\nu Y.\langle b\rangle Y) \wedge \langle a\rangle X)$
Q_1
$s_1, \mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X)$
$s_1, \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$
$s_1, (\nu Y.\langle b\rangle Y) \vee \langle a\rangle X$
$s_1, (\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X'$
Q_3
$s_1, \nu Y.\langle b\rangle Y$
$s_1, \langle a\rangle X$
$s_1, \nu Y'.\langle b\rangle Y'$
$s_1, \langle a\rangle X'$
$s_1, \langle b\rangle Y$
s_1, X
$s_1, \langle b\rangle Y'$
s_1, X'
s_2, Y
s_2, Y'
$s_2, \nu Y.\langle b\rangle Y$
$s_2, \nu Y'.\langle b\rangle Y'$
$s_2, \langle b\rangle Y$
Q_2
$s_2, \langle b\rangle Y'$
Q_4

Fig. 2. A partitioned game graph.

Let us fix the decomposition of the game graph shown in the previous proof and further helpful notions in the following definition:

Definition 2. *Let $\mathcal{T}$ be a labelled transition system and let φ be a formula of the alternation-free μ-calculus. Furthermore, let $\mathcal{G} = (Q, E)$ be their game graph.*

- *The* canonical decomposition *of $\mathcal{G}$ is the decomposition according to Theorem 2 into $Q_1, \ldots, Q_m$, which are tree-like ordered by $\leq$.*
- *The* escape configurations *of a component Q_i (denoted by $\lfloor Q_i \rfloor$ are the configurations which are in a child component and are successor configurations of a configuration in Q_i. That is:*

$$\lfloor Q_i \rfloor = \{q \in Q_j \mid Q_j \text{ is a child of } Q_i \text{ and } \exists q' \in Q_i \text{ such that } (q', q) \in E\}$$

- *The* component number *of a configuration of the game graph is the (unique) index i of a component that contains the configuration.*

It is obvious that every infinite play gets trapped within one Q_i. If the component is a ν-component, then ∀belard has lost the game. So he tries to leave a ν-component by reaching an escape configuration. Note that the second component of an escape configuration is labelled by a fixed-point formula and that $\lfloor Q_i \rfloor = \emptyset$ iff it is a leaf wrt. $\leq$.

The number of a component of a game graph's canonical decomposition is identical to the number of the component of the formula's graph according to Theorem 1. Even more, the component number of a configuration is identical to the number of the component of its formula label (which is defined in the obvious manner). Thus, once computed the component number of a (sub)formula as described in Section 2, it is a constant operation to check the component number of a configuration.

3 Sequential Model Checking

In the following, we restrict to the alternation-free μ-calculus. In this section, we present two sequential approaches for determining winning strategies, hereby solving the model-checking problem. The algorithms are designed in the way that they can easily be parallelised, which is carried out in the next section.

The basic idea of both algorithms is labelling a configuration q by *green* or *red*, depending on whether ∃loise or ∀belard has a winning strategy for the game starting in this configuration q. Furthermore, they both employ the canonical decomposition of the game graph (cf. Section 2). They differ in the order in which the several components are processed. The first algorithm proceeds bottom-up, the second top-down.

Colouring bottom-up First, let us discuss how to colour a single component. Let Q_i be a component of the canonical decomposition. To simplify the presentation, assume that Q_i is a μ-component. The forthcoming explanation can be dualised for ν-components. Let $\lfloor Q_i \rfloor$ denote its set of escape configurations and assume that every configuration in $\lfloor Q_i \rfloor$ is either labelled with *green* or *red* expressing that either ∃loise or ∀belard has a winning strategy from this configuration, resp. It is now obvious, that every play starting in a configuration of Q_i will either

1. eventually reach an escape configuration and never touch a configuration of Q_i again,
2. will end in a terminal configuration within Q_i, or
3. will go on infinitely within Q_i.

In the first situation, the winner is determined by the colour of the escape configuration. In the second case, the terminal configuration signalises whether ∃loise or ∀belard has won. The last case goes to ∀belard since a μ-formula is unwinded infinitely often.

The second case justifies colouring every terminal configuration within Q_i in the following way: If the formula component of the configuration is `true` or a box formula, then the configuration is coloured with *green*. Otherwise, when the formula component is `false` or a diamond formula, then the configuration is coloured with *red*.

Once a configuration $q \in Q_i \cup \lfloor Q_i \rfloor$ is labelled with *red* or *green*, its predecessors are labelled if possible: An $\bigwedge$-node q' is labelled with *red* if q is *red*, but labelled *green*, if all successors, i.e. q and all its neighbours, are *green*. An $\bigvee$-node is treated dually. If the predecessor has obtained a new colour, the labelling is propagated further. It is easy to see that, once a configuration obtained a colour, the colour is never changed.

Lemma 1. *The colouring process is terminating.*

However, the labelling process may leave some configurations of Q_i uncoloured. Let us understand that all remaining uncoloured configurations can be labelled with *red*.

Theorem 3. *For any game starting in a configuration without a colour, ∀belard has a winning strategy for a game starting in this configuration.*

Proof. First, check that every uncoloured configuration has at least one uncoloured successor configuration. ∀belard's strategy will be any choosing one uncoloured successor in this situation. Then he will win every play. Every uncoloured ∃loise-configuration has *red* or uncoloured successors, so ∃loise has the choice to move to configurations which are winning for ∀belard or to move to an uncoloured configuration. ∀belard will choose in an uncoloured configuration an uncoloured successor, or, if ∃loise has moved to a *red* configuration, he will choose a *red* successor. Summing up, every play will either end in a *red* terminal configuration, lead to a *red* escape configuration in which ∀belard has a winning strategy, or will go on infinitely often within Q_i and ∀belard wins.

The previous theorem is the crucial observation allowing a powerful parallel version of this algorithm. Unlike in many existing works on model checking this fragment, we do not use any cycle detection algorithm in the labelling process. We know that the described backward colour propagation process leaves only configurations uncoloured that are on or lead to a cycle which furthermore can be controlled by ∀belard.

The first sequential algorithm now processes the components in a bottom-up fashion. First, leaf components which have no escape configurations are considered and coloured. Now, for any parent component, the escape configurations are labelled and, again, our procedure will colour the component.

Let us turn back to our example shown in Figure 2. We have four components, $Q_1, \dots Q_4$. One leaf component is Q_2. The single terminal configuration $(s_2, \langle b \rangle Y)$ requires ∃loise to present a b-successor of s_2. However, in the underlying transition system, there is no successor. Thus, the configuration will be labelled with *red*. Propagating the colours to the predecessor configurations will colour every configuration of Q_2 with *red*.

The other leaf component Q_4 will be treated in the same manner as Q_2.

The next component to handle wrt. our tree order is Q_3. It has the single escape configuration $(s_1, \nu Y'.\langle b \rangle Y')$ which is already coloured with *red*. This colour is propagated to $(s_1, \nu Y'.\langle b \rangle Y' \wedge \langle a \rangle X')$ which now is coloured *red*. Further propagation will colour the whole component Q_3 with *red*.

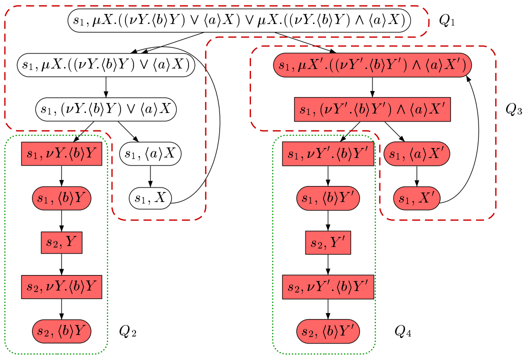

$s_1, \mu X.((\nu Y.\langle b \rangle Y) \vee \langle a \rangle X) \vee \mu X.((\nu Y.\langle b \rangle Y) \wedge \langle a \rangle X)$
Q_1
$s_1, \mu X.((\nu Y.\langle b \rangle Y) \vee \langle a \rangle X)$
$s_1, (\nu Y.\langle b \rangle Y) \vee \langle a \rangle X$
$s_1, \mu X'.((\nu Y'.\langle b \rangle Y') \wedge \langle a \rangle X')$
$s_1, (\nu Y'.\langle b \rangle Y') \wedge \langle a \rangle X'$
Q_3
$s_1, \nu Y.\langle b \rangle Y$
$s_1, \langle a \rangle X$
$s_1, \langle b \rangle Y$
s_1, X
s_2, Y
$s_2, \nu Y.\langle b \rangle Y$
$s_2, \langle b \rangle Y$
Q_2
$s_1, \nu Y'.\langle b \rangle Y'$
$s_1, \langle a \rangle X'$
$s_1, \langle b \rangle Y'$
s_1, X'
s_2, Y'
$s_2, \nu Y'.\langle b \rangle Y'$
$s_2, \langle b \rangle Y'$
Q_4

Fig. 3. Before the second phase

We have to proceed with Q_1. $(s_1, \nu Y.\langle b \rangle Y)$ propagates *red* to $(s_1, \nu Y.\langle b \rangle Y \vee \langle a \rangle X)$. Since the latter is an ∃loise-configuration, it remains uncoloured. A similar situation occurs for the propagation due to $(s_1, \mu X'.((\nu Y'.\langle b \rangle Y') \wedge \langle a \rangle X'))$. Thus, all colour information is propagated within Q_1. The current situation is depicted in Figure 3 in which *red* configurations are filled with 🟥. Now, the second phase of colouring a component comes into play. All remaining configurations will be labelled with *red* since Q_1 is a μ-component. Thus, ∀belard has a winning strategy for the presented game and we know that the underlying formula is not valid in the initial state of the transition system.

Complexity It is a simple matter to see that the previous labelling algorithm has a linear running time (in the worst case) wrt. the size of the game graph. The latter is bounded by the size of the underlying transition system (denoted by s) times the length of the formula (denoted by l). Hence, it is bounded by $s \times l$. However, only the part of the transition system *related* to the underlying formula has to be considered. For example, checking $\langle a \rangle \varphi$ in a state s requires only to look for a successor reachable by an action a that satisfies φ. All successors reachable by different actions need not be considered. While in the worst case, the whole transition system has to be considered checking a formula, only a part of the system has to be generated in typical examples. Thus, we can call our algorithm to be *local* or *on-the-fly*.

Colouring top-down For the second algorithm, assume that the game graph is again partitioned into components Q_i which form a tree. To consider as few components as possible, the algorithm will process the components in a top-down manner.

Again, let us first discuss how to colour a single component. Let Q_i be a component of the canonical decomposition. Again, we assume that Q_i is a μ-component recalling that the forthcoming explanation can be dualised for ν-components. Let $\lfloor Q_i \rfloor$ denote its escape configurations. However, we will *not* assume that every configuration in $\lfloor Q_i \rfloor$ is already coloured. Still, every play will either

1. eventually reach an escape configuration and never touch a configuration of Q_i again,
2. will end in a terminal configuration within Q_i, or
3. will go on infinitely within Q_i.

Again, the winner of a play is clear in Case 2. Furthermore, if ∃loise has neither a way to reach a winning terminal configuration, nor to leave the component she will loose. So, if she has no chance to reach a winning terminal configuration, the best we can hope for her, is that she indeed has a chance to leave the component successfully. The crucial point of our algorithm is that we initially colour all escape configurations of the component under consideration with *lightgreen* denoting that this configuration is probably a winning configuration for ∃loise.

As before, the colour information (full as well as light colours) is propagated to predecessor configurations and used for colouring it. That means, an $\bigwedge$-node is labelled with *red* if one successor is *red*, labelled with *lightred*, if no successor is *red* but at least one is *lightred*, labelled *lightgreen*, if all successors are *lightgreen* or *green*, and labelled with *green*, if all successors are *green*. In all other cases, the configuration remains unlabelled. A $\bigvee$-node is treated dually. Note that *lightred* comes only into play for ν-components. If the predecessor has got a new colour, the labelling is propagated further. A simple case analysis shows that once a configuration obtained a full colour, the colour is never changed. A light colour is only changed to the corresponding full colour.

Lemma 2. *The colouring process is terminating.*

Again, the labelling process may leave some configurations of Q_i uncoloured. Let us now understand, that all remaining uncoloured configurations can be labelled with *red*.

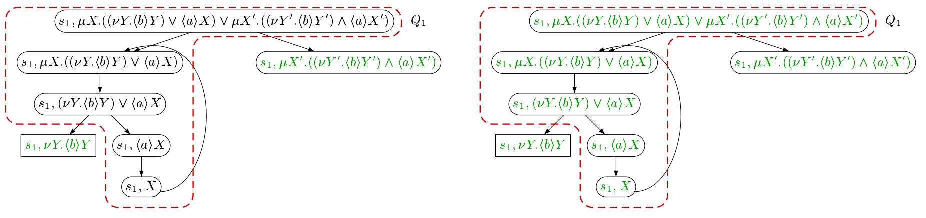
Theorem 4. *For any game starting in a configuration without a colour, ∀belard has a winning strategy for a game starting in this configuration.*

Proof. First, check that every uncoloured configuration has at least one uncoloured successor configuration. ∀belard's strategy will be any, choosing one uncoloured successor. Then he will win every play. Every uncoloured ∃loise-configuration has *red*, or uncoloured successors, so ∃loise has the choice to move to a configuration which is winning for ∀belard or to move to an uncoloured configuration. ∀belard will choose in an uncoloured configuration an uncoloured successor, or, if ∃loise has moved to a non-terminal *red* configuration, he will choose a *red* successor. Summing up, every play will either end in a *red* terminal configuration, move to a *red* escape configuration, in which ∀belard has a winning strategy, or will stay infinitely often within Q_i and ∀belard wins.

Still, the previous theorem is crucial for getting a powerful parallel algorithm.

Our component now may contain configurations which are coloured with *lightgreen*. However, we cannot guarantee that ∃loise has indeed a winning strategy for games starting in such a configuration. Thus, we remove the colour of such a configuration. If the initial configurations of the component are coloured, we are done. If not, we have to consider a child component to get further evidence.

Let us turn back to our example shown in Figure 2. We introduce the colour *white* to identify uncoloured configurations, assuming that initially every configuration has a *white* colour. We start with the root component Q_1. Both escape configurations are initially labelled with *lightgreen* (Figure 4(a)[3]).


$s_1, \mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$
Q_1
$s_1, \mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X)$
$s_1, \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$
$s_1, (\nu Y.\langle b\rangle Y) \vee \langle a\rangle X$
$s_1, \nu Y.\langle b\rangle Y$
$s_1, \langle a\rangle X$
s_1, X

(a) Starting with Q_1 (b) Propagating assumptions

Fig. 4. Two steps in the algorithm.

As shown in Figure 4(b), propagating the colour information will colour every configuration of Q_1 with *lightgreen*.

The subsequent phase of colouring *white*-configurations *red* and *lightgreen*-configurations *white* will turn the whole component to a complete *white* one so that the corresponding system looks similar to the one in the beginning. Thus, the assumptions did not help to find a winner. Therefore, we have to check a child component of Q_1. Let us proceed with Q_2. Since there are no escape configurations, the whole component is coloured as before. We learn that the *lightgreen* assumption for the initial configuration of Q_2 was too optimistic. Redoing the colouring of Q_1, now with more but still not full information, will colour some configurations of Q_1 *red* but will still leave the initial configuration uncoloured. Figure 5(b) shows the coloured game graph right before recolouring the assumed coloured *lightgreen* back to *white*.

We turn our attention to Q_3. We assume that ∃loise has a chance to leave the component via $(s_1, \nu Y'.\langle b\rangle Y')$. Thus, we colour this configuration *lightgreen*.

[3] *lightgreen* configurations are identified by writing their label in the following style: xxx

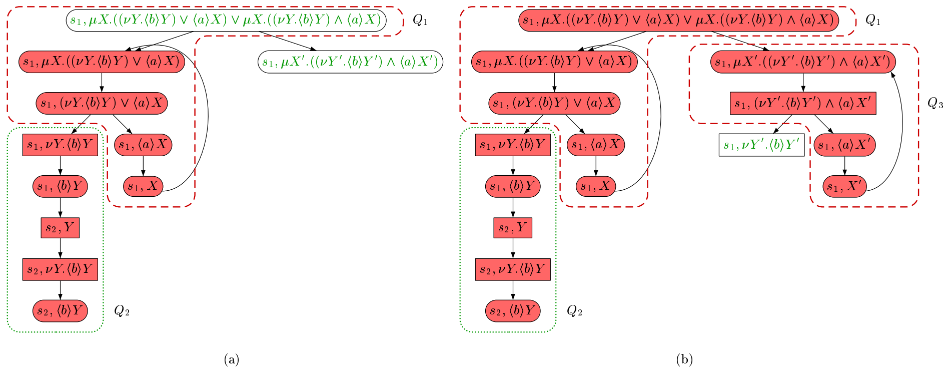


Fig. 5. Subsequent steps in the algorithm.

However, the propagation does not influence the preceding $\bigwedge$-node. So all remaining configurations are coloured *red*. Now, all escape configurations of Q_1 are coloured and a further colouring process will colour the complete component Q_1 *red*. Note that we saved the time and especially space for considering Q_4. Figure 5(b) shows the coloured game graph, again right before recolouring the assumed coloured *lightgreen* back to *white*.

Complexity It is a simple matter to see that the previous labelling algorithm has a running time bounded by $n \times m$ where n is the number of configurations and m is the size of the maximum length of a path from the root component to a leaf component. The latter number is bounded by the nesting of fixed-point formulae, which is at most the length of the formula. Thus, we get as an upper bound $s \times l^2$ where s is the size of the underlying transition system and l is the length of the formula. While in the worst case, this complexity is worse than in the bottom-up approach, we found out that the algorithm often detects the truth-value of a formula in a given state much faster. Note, that this algorithm behaves *even more on-the-fly* than the previous one.

4 Parallel Model Checking

Given a transition system and an L^1_μ-formula, our approach is both to construct the game graph as well as to determine the colour of its nodes in parallel. The idea of our parallel algorithm is that all processors are working in parallel on one component, whereas the components are treated one-by-one.

Distributing the game graph We employ a somehow standard approach distributing and constructing a (component of the) game graph in parallel [20, 4, 5]. As a data structure, we employ adjacency lists. We need also links to the predecessor as well as to the successor of a node for the labelling algorithm. A component is constructed in parallel by a typical breadth-first strategy. Given a node q, determine its successors $q_1, \ldots, q_n$. To obtain a deterministic distribution of the configurations over the workstation cluster, one takes a function in the spirit of a hash function assigning to every configuration an integer and subsequently its value modulo the number of processors. This function f determines the location of every node within the network uniquely and without global knowledge. Thus, we can send each $q \in \{q_1, \ldots, q_n\}$ to its processors $f(q)$. If q is already in the local store of $f(q)$, then q is reached a second time, hence the procedure stops. If predecessors of q were sent together with q, the list of predecessors is augmented accordingly. If q is not in the local memory of $f(q)$, it is stored there together with the given predecessors as well as all its successors. These are sent in the same manner to their (wrt. f) processors, together with the information that q is a predecessor. The corresponding processes update their local memory similarly.

Please consult [4] for a thoroughly discussion of this and other possible approaches storing distributed transition systems.

Labelling the game graph As many of the existing model-checking algorithms that we are aware of, use cycle detection algorithms, which are unlikely to be parallelised in a simple way, we extend our sequential algorithms described in Section 3 towards a parallel implementation. As explained in the previous paragraph, it is easy to construct (a component of) the game graph in parallel employing a breadth-first search. When a terminal configuration is reached, a backwards colouring process can be initiated as described in Section 3. This can be carried out in parallel in the obvious manner. If all colour information is propagated, the sequential algorithm performs a colouring of uncoloured nodes and an erasing of light colours (cf. Section 3). It is no problem to do this recolouring on the distributed game graph in parallel. Thanks to Theorem 3 or Theorem 4, no cycle detection is necessary but every workstation can to this recolouring step on its local part of the game graph.

However, to check that all colour information has been propagated, a distributed termination-check algorithm is employed. We use a token termination algorithm due to [9]. It has the advantage that it scales well wrt. the number of workstations and that its complexity is independent of the size of the game graph. The components may be labelled in a bottom-up or top-down manner as described in Section 3. Thus, we get a set of algorithms differing in the order the components are processed.

The algorithm To describe our approach in more detail, we show several fragments of our algorithm in *pseudo code*. Especially, we note that the two steps of constructing the game graph and labelling the nodes can be carried out in an integrated way. The most important function is shown in Figure 6(a). Given a component number, it expands all nodes of the component. It can be applied either for a parallel bottom-up or top-down labelling algorithm. Since the colour information of a terminal node is always a correct (full) colour, a colouring process is initiated, if a terminal configuration is reached.

```
Function processSuccs(node) // compute and expand succs
  succs ← computeSuccs(node.conf)
  for s ∈ succs do
    sendMessageTo (Expand s node.conf) f(s)
  succss ← [ ( suc.conf ← s , suc.colour ← white ) | s in succs ]
  node.succs ← succss
end

Function expandComp(comp)
  for node in graph[comp].initialNodes // start with initial nodes
    processSuccs(node)
  until hasTerminated do
    msg ← readMessage;
    case msg of
      Expand conf pred:
        if lookupGraph(conf, node) ≠ fail then // node already visited
          if node.colour ≠ white and pred ∉ node.preds then
            sendMessageTo (Colour pred node.conf colour) f(pred)
          addPreds(node.preds, pred)
        else
          node ← newNode, node.conf ← conf, node.pred ← pred
          if isTerminal(node) then
            node.colour ← InitialColor (node.conf)
            sendMessageTo (Colour pred node.conf colour) f(pred)
          else  // new node
            node.colour ← white
          if not (isTerminal(node) or (IsEscapeConf(comp,s))) then
            processSuccs(node)
      Colour conf child colour:
        lookupGraph(conf, node)
        updateSucc(node.succs, child, colour)
        newcolour ← computeColour(node)
        if newcolour ≠ oldcolour then
          node.color ← newcolour
          for p ∈ preds do sendMessageTo (Colour p conf newcolour) f(p)
end
```

(a) Expanding a component

```
Function recolourComp(comp)
  case type(comp) of
    μ : colour := red
    ν : colour := green
  for node in graph[comp]
    if node.color = white then
      node.color := colour
    if node.color in
        {lightred, lightgreen} then
      node.color := white
end
```

(b) Recolouring

```
Function computeColour(node,succs)
begin
  case
    node is ⋁−node:
      case
        all (= Red) succs: Red
        any (= Green) succs: Green
        else: White
    node is ⋀−node:
      case
        all (= Green) succs: Green
        any (= Red) succs: Red
        else: White
end
```

(c) Computing the colour

Fig. 6. The algorithms

We assume that the game graph is represented as a list of nodes where each node is a record containing the label of the current configuration, its colour, and a list of predecessors and successors. Furthermore, we assume that initial configurations are already stored in the graph when calling the function expandComp. The latter are first expanded (lines 10–11). Then, the function enters a message loop awaiting either expand or colour requests. If a configuration should be expanded, it is checked whether the configuration has already been processed (16). If so, a possible new predecessor of this configuration is stored. Furthermore, the predecessor is informed about the colour of the current node, if it has been coloured already. Otherwise, the node is added to the graph, and the configuration and the predecessor are stored (21). A colouring process is initiated, if the current node is a terminal one (22–26). Furthermore, if the node is neither terminal or an escape configuration, it successors are are considered (28). Thus, its successors are computed and expanded (2–4). Furthermore, the successors together with an initially white-colour are stored in the current node (5–6). Colour informations are processed in the expected manner (29–35).

The function expandComp is the main ingredient for building a complete algorithm. If one is interested in a bottom-up algorithm, one can call expandComp in a depth-first manner for the components of the game-graph. Then, starting from the leafs, the function recolourComp (Figure 6(b), *light* colours are only present in the top-down version) can be called in a bottom-up manner. Of course, after processing a component, a colour propagation process for the initial nodes of the component has to be initiated before recolouring the next component.

For the bottom-up algorithm, the colour can be computed as described in Figure 6(c).

For the top-down colouring version of our algorithm the expandComp function is called first. Then, a colouring process with light colours is initiated starting from the escape configurations of the component. Now, a recolour process (Figure 6(b)) is started.

There are several possibilities to process the components. In the examples shown in Section 3, we suggested a depth-first strategy. However, one could also use a breadth-first, bounded depth-first, or parallel breadth-first strategy. Depending on the employed strategy, the run-time of our algorithm is linear or quadratic. Note, that the space required by our algorithm is linear in the size of the game graph. The employment of light colours might save considering a significant part of the game graph but may also augment the runtime, if the whole game graph has to be considered.

Theorem 5. *The algorithms described before label a node (s, ψ) of the game graph with green if $\mathcal{T}, s \models \psi$. Otherwise, the node is labelled with red.*

Variations of the algorithm We already mentioned that there are several possibilities to process the components. Note that only the colour of the escape configuration is needed when colouring a component. Thus, all other nodes of a child component can be deleted for colouring the current component. In general, it is possible to formulate the algorithm in the way that only a *single* component plus some control information is stored in the workstation cluster at the same time certainly lifting the limits for systems to be model checked in practice. For lack of space, we do not provide the details.

Another variant of our algorithm can be obtained by taking subformulae instead of the occurrence set of the subformulae for defining the graph of the formula (Definition 1). The resulting effect will be that the components of the game graph no longer constitute a natural tree order but form a directed acyclic graph which is no longer necessarily a tree. Although the resulting game graph can be expected to be smaller, the tree order simplifies the decision when a component can be removed, as described in the previous paragraph.

Winning strategies As pointed out already when motivating the use of games, the winning strategy does not only provide an answer to the model-checking question but can also be applied for interactively debugging the underlying system. It is easy to extend our algorithm towards providing a winning strategy. Note that the colour of a terminal node in the game graph is a winning position for one of the players. If the colour information is propagated to a predecessor without a colour (*white*) and this leads to a colour of the predecessor, it is clear how the corresponding winner has to chose. In other words, when a node gets a colour because of a `(Colour conf succ colour)` message, the strategy is choosing `succ` in configuration `conf`. If a node is coloured in the function recolourComp, we pointed out in the proof of Theorem 3 and Theorem 4 that the right strategy is choosing a previously *white* successor.

5 Experimental Results

We have tested our approach within our verification tool Truth [17] implemented in Haskell as well as with a stand-alone version written in C++.

The Haskell version We implemented the distribution routine on its own as well as the combined iterative labelling routine described in the previous section. As implementation language we have chosen the purely functional programming language Haskell allowing us to embed this algorithm in the verification tool Truth and to prototype a *concise* reference implementation. The actual Haskell source code of the algorithm has less than 280 lines of code. As the distribution routine is the same as in [5], we refer to this paper for the positive results we obtained using this distribution mechanism. Figure 7 shows the measured runtime results of the state distribution integrated with the parallel labelling algorithm for a single component on a NOW consisting of up to 52 processors and a total of 13GB main memory, which are connected with a usual 100MBit Fast-Ethernet network.

Our approach also scales very well with regard to the overall runtime (Figure 7). Note that, because of the size of the game graphs we inspected, we did not get results when running the algorithm on less than five workstations due to memory restrictions. Therefore, the shown speedups are calculated relative to 5 processors instead of one. We found that we gain a linear speedup for reasonably large game graphs (in fact, for graphs with more than 500.000 states, we even got *superlinear* speedups, which we will discuss later). The results are especially satisfying, if one considers that—for reasons of simplicity—we did not try to employ well-known optimisation means, for example reducing the communication overhead by packing several states into one message.

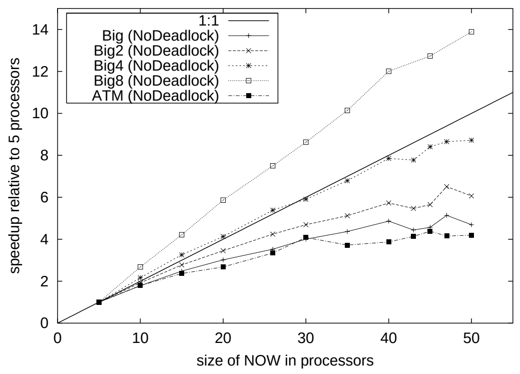


Fig. 7. runtime results

Also, due to our choice of Haskell as implementation language and its inherent inefficiency, we did not focus on optimising the internal data structures either. We use purely functional data structures like balanced trees and lists rather than destructively updateable arrays or hash tables. This is also the reason for the superlinear speedups we mentioned before. We found that the overhead for insertions and lookups on our internal data structures dramatically increases with the number of stored states. We verified this by running all processes on a *single processor* in parallel and replacing the network message passing with inter-process communication. The expected result would have been to find similar runtimes as one process would achieve in this situation, or even slightly worse due to operating-system context switches between the processes running in parallel. However, we found that there is a significant speedup, because the internal data structures are less crowded and therefore lookups and insertions are considerably cheaper.

The C++ version We have implemented a simple prototype tailored for transition systems computed by the μCRL tool set [3]. We have tried the version on a transition system with 13 million states. We were able to check a mutual exclusion property within 9 minutes on the NOW. Note that existing model checkers for μCRL failed to show the property due to memory restrictions. However, we have to learn that the algorithm does not scale as good as the Haskell version. As soon as the transition system fits into the accumulated memory of the computers, further computers provide no significant speed-up. The reason is that, in contrast to the Haskell version where the transition system is computed on-the-fly from a given system of CCS process equations, the system is already computed in the C++ version, a fact which is not used by our algorithm. Thus, every computer is mainly concerned with labelling the game graph and sending colour information to the other computers, and communication is a costly operation within a NOW. It is therefore important to interleave the computation of the transition system together with the computation of the model-checking algorithm. Note that it took 3 hours to produce the mentioned transition system with the μCRL tool set before we were able to check the system within 9 minutes with our tool.

Since our C++ version is just a simple prototype and provides a lot of optimisation possibilities, we currently work on a sophisticated version of our parallel model checker, which can be applied for getting further insights to the run-time behaviour of our approach. This will be integrated with the routines of μCRL for generating a transition system, so that we will get a parallel on-the-fly model checker for μCRL.[4]

6 Conclusion

In this paper, we have presented a *parallel* game-based model-checking algorithm for an important fragment of the μ-calculus. The demand for parallel algorithms becomes visible by considering the memory and run-time consumptions of sequential algorithms. Since the employed fragment of the μ-calculus subsumes the well-known logic CTL, it is of high practical interest. We have implemented a prototype of our approach within the verification platform TRUTH. We found out that the algorithm scales very well wrt. run-time and memory consumption when enlarging the NOW.

With our parallel algorithm, answers are computed more quickly, and, more importantly, there are numerous cases in which the sequential algorithm fails because of memory restrictions and the parallel version is able to check a formula. From the practical point of view, it is a central feature of a verification tool to give an answer in as many cases as possible. Thus, a decent implementation of this algorithm will be carried out to get further practical results.

References

1. H. R. Andersen. Model checking and Boolean graphs. *Theoretical Computer Science*, 126(1):3–30, 11 Apr. 1994.
2. S. Basonov. Parallel implementation of BDD on DSM systems. Master's thesis, Computer Science Department, Technion, 1998.
3. S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the 13th Conference on Computer-Aided Verification (CAV'01)*, *LNCS* 2102, p. 250–254. Springer, July 2001.
4. B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation free μ–calculus. Technical Report AIB-04-2001, RWTH Aachen, 03/2001.
5. B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In T. Margaria and W. Yi, editors, *Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, *LNCS* 2031, p. 543–558. Springer, Apr. 2001.
6. L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model-checking based on negative cycle detection. In *Proc. of 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, LNCS. Springer, Dec. 2001.

[4] This is also the reason why we did not optimise our algorithm wrt. an a-priori-known transition systems.

7. G. Cabodi, P. Camurati, and S. Que. Improved reachability analysis of large FSM. In *Proc. of the IEEE International Conference on Computer-Aided Design*, p. 354–360. IEEE Computer Society Press, June 1996.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* The MIT Press, Cambridge, Massachusetts, 1999.
9. E. W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, June 1983.
10. E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
11. E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In C. Courcoubetis, editor, *Proc. 5th International Computer-Aided Verification Conference*, *LNCS* 697, p. 385–396. Springer, 1993.
12. O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the 13th Conference on Computer-Aided Verification (CAV'01)*, of *LNCS* 2102, p. 350–362. Springer, July 2001.
13. H. Hiraishi, K. Hamaguchi, H. Ochi, and S. Yajima. Vectorized symbolic model checking of computation tree logic for sequential machine verification. In K. G. Larsen and A. Skou, editors, *Proc. of Computer Aided Verification (CAV '91)*, *LNCS* 575, p. 214–224, Berlin, Germany, July 1992. Springer.
14. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, Dec. 1983.
15. 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.
16. M. Leucker. Model checking games for the alternation free mu-calculus and alternating automata. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proc. of the 6th International Conference on Logic for Programming and Automated Reasoning "(LPAR'99)"*, *LNAI* 1705, p. 77–91. Springer, 1999.
17. M. Leucker and T. Noll. Truth/SLC - A parallel verification platform for concurrent systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of the 13th Conference on Computer-Aided Verification (CAV'01)*, *LNCS* 2102, p. 255–259. Springer, July 2001.
18. A. Mader. *Verification of Modal Properties Using Boolean Equation Systems.* PhD thesis, Technische Universität München, 1996.
19. A. A. Narayan, J. J. J. Isles, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned–roBBDs. In *Proc. of the IEEE International Conference on Computer-Aided Design*, p. 388–393. IEEE Computer Society Press, June 1997.
20. U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In O. Grumberg, editor, *Computer-Aided Verification, 9th International Conference*, *LNCS* 1254, p. 256–267. Springer, June 1997. Haifa, Israel, June 22-25.
21. C. Stirling. Games for bisimulation and model checking, July 1996. Notes for Mathfit Workshop on finite model theory, University of Wales, Swansea,.
22. A. L. Stornetta. Implementation of an efficient parallel BDD package. Master's thesis, University of California, Santa Barbara, 1995.
23. S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *Proc. of the 9th Annual IEEE Symposium on Logic in Computer Science*, p. 154–163, Paris, France, 4–7 July 1994. IEEE Computer Society Press.