RWTH Aachen

Department of Computer Science
Technical Report

Local Parallel Model Checking for the Alternation Free μ-Calculus

Benedikt Bollig, Martin Leucker, and Michael Weber

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2001-04

RWTH Aachen · Department of Computer Science · March 2001

Local Parallel Model Checking for the Alternation Free μ–Calculus

Benedikt Bollig, Martin Leucker, and Michael Weber*

Lehrstuhl für Informatik II, RWTH Aachen, Germany
{bollig, leucker, michaelw}@informatik.rwth-aachen.de

Abstract. In this paper, we describe the design and implementation 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 model checking problem for this fragment in terms of two–player games. Our algorithm determines the corresponding winner in parallel. Furthermore, a winning strategy is computed which may be employed for interactively debugging the underlying system. The algorithm is designed to run on a network of workstations. Depending on the chosen variant, its complexity is linear or quadratic. A prototype implementation within the verification tool Truth shows promising run–time and memory behaviour in practice.

1 Introduction

Model checking is becoming more and more popular for the verification of complex hardware and software systems. These systems are usually given by a formal description which can be transformed into a transition system. A desired property of the system, on the other hand, is usually specified as a formula of a temporal logic. A model checking algorithm answers the question whether the (transition) system satisfies this property. Numerous case studies have shown that this approach improves the early detection of errors during the design process (see [8] for an overview).

Despite the developments in the last years, the so–called *state space explosion* still limits its application. While *partial order reduction* [32] or *symbolic model checking* [28] reduce the state space by orders of magnitude, typical verification tasks still last days on a single workstation or are even (practically) undecidable due to memory restrictions (see for example [15]).

On the other hand, cheap yet powerful parallel computers can be constructed of Networks Of Workstations (*NOW*s). Various message–passing–interfaces such as MPI [13] or PVM [14] and their corresponding implementations (MPICH[1], LAM[2]) provide high–level parallel abstractions to application programmers and thus enables them to develop portable and efficient parallel programs. From the outside, a NOW appears as one single parallel computer with high computing

* The work of this author was supported by the "Graduiertenkolleg für Software für Kommunikationssysteme

[1] http://www-unix.mcs.anl.gov/mpi/mpich/

[2] http://www.mpi.nd.edu/lam/

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 a fundamental goal 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 famous logic for expressing specifications is Kozen's μ–calculus [21], a temporal logic offering boolean combination of formulas and, especially, labelled *next*–state, minimal, and maximal fixpoint quantifiers. It is the most expressive logic adequate for transition systems representing concurrent systems since it is expressive complete with respect to the fragment of second order logic consisting of the formulas not distinguishing *bisimilar* transition systems [20]. One of the most important unanswered questions about the μ–calculus is whether there is a polynomial time model checking algorithm improving the known bounds viz NP and CoNP [11].

For practical applications, however, it suffices to restrict the μ–calculus in order to gain tractable model checking procedures. The alternation free fragment, denoted by L^1_μ, prohibits the nesting of minimal and maximal fixpoint operators. It allows the formulation of many *safety* as well as *liveness* properties. While this fragment is already important on its own, it subsumes the logic CTL [12] 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, and several sequential model checking procedures are given in the literature [9, 1, 22, 3]. 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. [23] compares the algorithms in detail.

Before starting to think about a concrete algorithm, we should consider its limitations, i.e. its complexity. In complexity theory, it is a well–accepted view that problems within the class NC admit a promising parallel computing algorithm. NC is based on the Boolean circuit model for computation and describes the problems computable in polylogarithmic time with polynomially many processors. It can be shown that NC is contained in P. Problems outside of NC are consequently considered to be inherently sequential. However, it is not known whether NC=P. If not, then especially P–complete problems cannot be in NC. Hence, we call P–complete problems *inherently sequential* [31].

We show that model checking L^1_μ is inherently sequential, limiting our enthusiasm for finding a (theoretically) good parallel model checking algorithm. All we can hope is to find a linear time algorithm (unless NC equals P). We present a parallel local model checking algorithm and several of its variations which have linear or quadratic time complexity. We have implemented the algorithm within our verification tool TRUTH [24] and we found out that it behaves very well for many practical problems.

Our algorithm is based on a characterisation of the model checking problem for this fragment in terms of two–person games due to Stirling [35]. Strictly speaking, we present a parallel algorithm for colouring so–called game graphs corresponding to the underlying model checking problem. This colouring answers the model checking problem. Furthermore, we explain that our algorithm can be extended to compute winning strategies without further costs. A strategy may be employed by the user of a verification tool for debugging the underlying system interactively [35].

A different characterisation of the model checking problem can be given in terms of so–called 1–letter–simple–weak–alternating–Büchi automata [22]. However, these are related to games in a straightforward manner [26]. Hence, our algorithm can also be understood as a parallel procedure for checking the emptiness of this kind of automata.

Indeed, the first variation of our parallel algorithm is very similar to a solution of the model checking problem described in [22]. However, the proposed algorithm employs the detection of cycles which is unlikely to be parallelised in a simple way. The key observation is that we can omit this step by employing structural information about the underlying graph. This yields a simple and efficient parallel algorithm. Furthermore, we present a second version of our algorithm that provides an extended on–the–fly behaviour for a cheap price.

Until today, not much effort has been taken to consider parallel model checking algorithms. [36, 2] present parallelised data structures which employ further computers within a network as a substitute for external storage. The algorithms described in [30, 6] 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. Stern and Dill [34] show how to carry out a parallel reachability analysis. The distribution of the underlying structure is similar to the one presented here. But their algorithm is not appropriate for model checking temporal logic formulas. [18] presents a parallel reachability analysis algorithm for BDDs. They argue that many safety properties can be formulated as a reachability problem. In this way, their algorithm allows checking safety formulas. However, *liveness* properties which can be expressed within L^1_μ are not supported. Furthermore, [19] argues that explicit state representation as well as BDDs have application domains in which they outperform the other one. Moreover, BDD–based algorithms generally do not provide counter examples which are important in practice. In [5], we presented a less powerful parallel algorithm for (a fragment) the same problem. [17] introduced a symbolic parallel algorithm for the full μ–calculus. However, it is global and requires the transition system to be constructed totally. Our main contribution is the first *local* parallel model checking algorithm for L^1_μ that supports interactive debugging.

After fixing some basic notations in the next section, we introduce the syntax and semantics of the μ–calculus in Section 3. Model Checking for this logic is defined in Section 4 and it is shown that this problem is *inherently sequential* for L^1_μ. In Section 5, 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 sequential and parallel algorithms. To simplify our presentation, we start in Section 6 with the presentation of sequential model checking algorithms that admit a simple parallel version. The corresponding parallel model checking procedure is shown in Section 7. Before we draw the conclusion of our approach, we present experimental results in Section 8.

2 Preliminaries

Let us fix some order and graph theoretic notions and notations.

A tree order is a tuple $(Q, \leq)$ such that $\leq$ is a partial ordering relation on Q and its covering relation (or its Hasse diagram) is a tree. More specifically, assume $\leq$ is a reflexive, antisymmetric, and transitive relation. The *covering relation* $\lessdot$ of $\leq$ is defined by $\lessdot :=\leq - \leq^2$. Instead of $(q, q') \in \lessdot$ we also write $q \lessdot q'$. We call $\leq$ a *tree order* iff

- there is a unique $q \in Q$ such that $q \leq q'$ for all $q' \in Q$, and
- for all $q, q', \hat{q} \in Q$, $q \lessdot \hat{q}$ and $q' \lessdot \hat{q}$ implies $q = q'$.

Notions of parents and children for elements of Q with respect to $\leq$ correspond to the usual ones for elements of Q with respect to $\lessdot$.

A (directed) *graph* $\mathcal{G}$ is a tuple $\mathcal{G} = (Q, \rightarrow)$ where Q is an arbitrary set and $\rightarrow \subseteq Q \times Q$.[3] The elements of Q are called *nodes* of $\mathcal{G}$ and the elements of $\rightarrow$ are called edges of $\mathcal{G}$. For $q, q' \in Q$, a path from q to q' is a sequence of nodes $q_0, \ldots, q_n \in Q$ such that $q = q_0 \rightarrow \ldots \rightarrow q_n = q'$. A *cycle* in $\mathcal{G}$ is a sequence of nodes $q_0, \ldots, q_n$ such that $q_0 \rightarrow \ldots \rightarrow q_n \rightarrow q_0$. We say that a node $q \in Q$ *is contained* or *reached in* a cycle iff there is a cycle $q_0, \ldots, q_n$ in $\mathcal{G}$ and $q = q_i$ for an $i \in \{0, \ldots, n\}$.

A *component* of a graph $\mathcal{G}$ is a subgraph $\mathcal{G}'$ of $\mathcal{G}$ *induced by* a set of nodes Q', that is $\mathcal{G}' = (Q', \rightarrow')$, $Q' \subseteq Q$, and $\rightarrow' = \rightarrow \cap (Q' \times Q')$. Sometimes, we call also Q' a component. A *(strongly) connected component* is a component $\mathcal{G}' = (Q', \rightarrow')$ of $\mathcal{G}$ such that for all $q, q' \in Q'$ there is a path from q to q' in $\mathcal{G}'$. A (connected) component and a cycle are called *non–trivial* if they contain a least two nodes.

Let $\mathcal{G}' = (Q', \rightarrow')$ and $\mathcal{G}'' = (Q'', \rightarrow'')$ be two disjoint components of $\mathcal{G}$, i.e. $\mathcal{G}'$ and $\mathcal{G}''$ are components of $\mathcal{G}$ such that $Q' \cap Q'' = \emptyset$. Furthermore, assume that there is no edge from a node in Q'' to a node in Q', thus, $\rightarrow \cap (Q'' \times Q') = \emptyset$. Then every edge from a node $q' \in Q'$ to a node $q'' \in Q''$ ($q' \rightarrow q''$) is called a *bridge*.

In the following sections, we consider node labelled graphs, especially graphs, where the nodes are labelled by formulas. We say that a cycle contains a formula φ, if the cycle contains a node that is labelled by φ.

[3] We only consider *directed* graphs in this paper.

3 The μ–Calculus

In this section, we recall the syntax and semantics of the (modal) μ–calculus and its alternation free fragment.

3.1 Syntax and Semantics

Let *Var* be a set of fixpoint variables, *Prop* a set of propositional variables, and $\mathcal{A}$ a set of actions. We require the fixpoint variables to be distinct from the propositional variables ($Var \cap Prop = \emptyset$). Formulas of the propositional modal μ–calculus over *Var* and $\mathcal{A}$ in positive form as introduced by [21] are defined as follows:

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

where $X \in Var$, $p \in Prop$, and K ranges over subsets of actions $\mathcal{A}$. Like [35], we allow sets of actions instead of single actions appearing in modalities.[4]

For a formula of the μ–calculus, we introduce the notion of *subformulas*, *free*, and *bound* variables as usual. We call a formula a *sentence* iff it contains no free fixpoint variable. Furthermore, we say that φ is a μ–formula iff $\varphi = \mu X.\psi$ for appropriate X and ψ. ν–formulas are introduced in a similar vein. From now on, we assume all formulas to be sentences.

A formula φ is *normal* if 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 if it identifies a ν–formula, and μ–variable otherwise. In the following, we assume all formulas to be normal.

Let $\mathcal{T} = (S, T, \mathcal{A}, s_0, P)$ be a *labelled transition system* 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 $s \stackrel{a}{\to} t$ instead of $(s, a, t) \in T$. Furthermore, let $s_0 \in S$ be the *initial state* of the transition system and let P denote a function $P : Prop \to 2^S$ yielding the the set of states in which a given proposition holds. We employ valuations V mapping a fixpoint variable X to a set of states $V(X) \subseteq S$. Let $V[X/E]$, $E \subseteq S$, be the valuation which is the same as V except for X where $V(X) = E$. Given a labelled transition system $\mathcal{T} = (S, T, \mathcal{A}, s_0, P)$ and a formula φ over *Var*, *Prop*, and $\mathcal{A}$, the satisfaction of φ with respect to $\mathcal{T}$ and a state $s \in S$ is defined for `true`, `false`, disjunction, and conjunction in the usual way, the semantics of a propositional variable yields the set of states in which it holds, thus, $P(p)$, and for the temporal and fixpoint operators as follows:

$\mathcal{T}, s \models_V [K]\varphi$ iff $\forall a \in K$: if $s \stackrel{a}{\to} t$ then $\mathcal{T}, t \models_V \varphi$
$\mathcal{T}, s \models_V \langle K \rangle \varphi$ iff $\exists a \in K$: $s \stackrel{a}{\to} t$ and $\mathcal{T}, t \models_V \varphi$
$\mathcal{T}, s \models_V \mu X.\varphi$ iff $\forall E \subseteq S$: if $s \notin E$ then $\exists t \in S : t \notin E$ and $\mathcal{T}, t \models_{V[X/E]} \varphi$
$\mathcal{T}, s \models_V \nu X.\varphi$ iff $\exists E \subseteq S, s \in E$: and $\forall t \in E : \mathcal{T}, t \models_{V[X/E]} \varphi$

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

We abbreviate $\mathcal{T}, s \models_V \varphi$ by $\mathcal{T}, s \models \varphi$ if V is known from the context and $\mathcal{T}, s_0 \models_V \varphi$ by $\mathcal{T} \models \varphi$. We use identifiers like $\varphi, \psi, \dots$ for formulas, $s, t, \dots$ for states, and $a, b, \dots$ for actions of the transition system under consideration. K denotes a set of actions. Whenever the sort of the fixpoint does not matter, we use σ for either μ or ν. Similarly, we write $\llbracket K \rrbracket$ for $\langle K \rangle$ or $[K]$, $\star$ for $\vee$ or $\wedge$, and $\#$ for $\sigma X.$ or $\llbracket K \rrbracket$.

Looking again at the semantics of our formulas we observe that there is no real difference between free fixpoint variables and propositional variables. One could likewise consider every free variable to be a propositional one. However, notions of sentences etc. would be more technical.

Essential for our further development is a formula's graph representation. To simplify the definition, let us first introduce the tree representation of a formula in the usual manner. 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 $\pi \in Occ(\varphi_i)$.
- $1\pi \in Occ(\varphi)$, if $\varphi = \#\varphi_1$ and $\pi \in Occ(\varphi_1)$.

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 now can assign to every φ a $Sub(\varphi)$–labelled tree with nodes $Occ(\varphi)$ and edge set $\rightarrow$ defined by

$$\rightarrow = \{(\pi, i\pi) \mid \pi, i\pi \in Occ(\varphi), \pi \in \mathbb{N}^*, i \in \mathbb{N}\}$$

The labels are assigned in the expected manner by $\lambda(\pi) = \varphi|_\pi$. Altogether, $\mathcal{T}(\varphi) = (Occ(\varphi), \rightarrow, \lambda)$ 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 with edges from fixpoint variables back to the fixpoint formula it identifies.

Definition 1. *Let φ be a formula of the μ–calculus. Let $\mathcal{T}(\varphi) = (Occ(\varphi), \rightarrow, \lambda)$ be its tree representation. The* graph *of φ denoted by $\mathcal{G}(\varphi)$ is the tuple $(Occ(\varphi), \rightarrow', \lambda)$ where*

$$\rightarrow' = \rightarrow \cup \{(\pi, \pi') \mid \lambda(\pi) = X \textit{ and } \lambda(\pi') = \sigma X.\varphi' \textit{ for } X \in Var \textit{ and approp. } \varphi'\}$$

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.

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

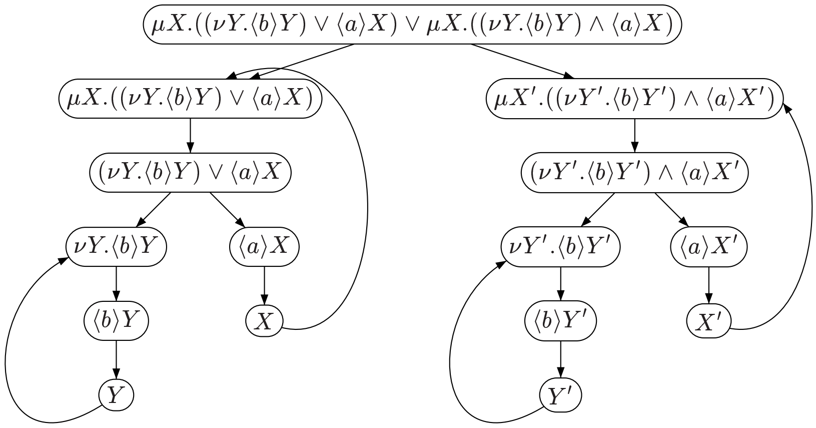
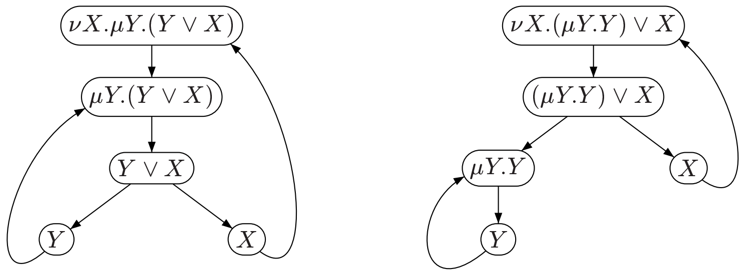


Fig. 1. The graph for $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$.



(a) alternation

(b) alternation-free

Fig. 2. Alternation $(\nu X.(\mu Y.Y \vee X))$ vs. alternation-free $(\nu X.(\mu Y.Y) \vee X)$

no cycle with a ν–variable as well as with a μ–variable. Figure 2(a) shows the graph of an alternating formula which has a cycle containing X as well as Y. Contrary, Figure 2(b) shows the graph of an alternation–free formula which has two maximal strongly connected components, one on which X occurs, a second containing Y.

For the rest of this section we restrict to the alternation free fragment of the μ–calculus.

A simple but essential observation is that the graph of a formula can naturally be decomposed into maximal strongly connected components and trees with certain features (cf. Figure 3).

Lemma 1. *Let φ be a formula and let $\mathcal{G}(\varphi) = (Q, \rightarrow, \lambda)$ be its graph. Then there exists a collection of $Q_1, \dots, Q_m$ such that the following holds:*

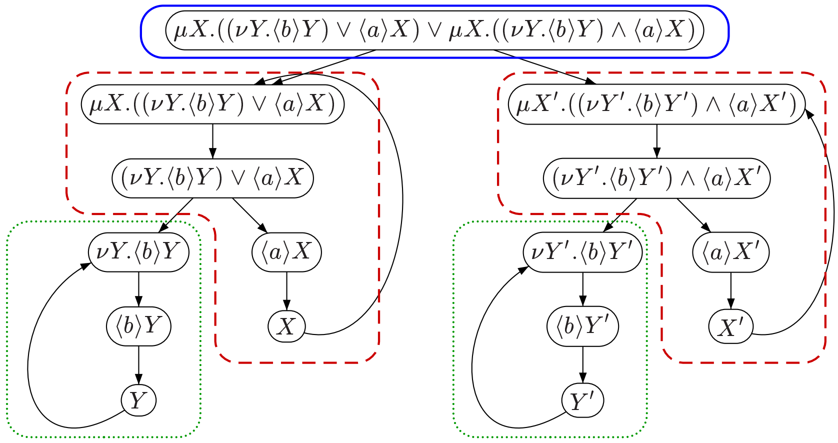


Fig. 3. The partition of the graph for $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$.

1. *The collection of the Q_i is 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 $Q_i \cap Q_j = \emptyset$.*
2. *Every subgraph induced by Q_i is exactly one of*
   (a) a non–trivial maximal strongly connected component in which only either μ–formulas or ν–formulas occur.
   (b) a tree.
3. *There exists a tree order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ with an edge from q to q', we have $Q_i \leq Q_j$. More specifically, edges from a node in Q_i lead to configurations in either the same Q_i or a child Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$.*

Proof. First, consider the nodes of maximal non–trivial strongly connected components. Cycles can only occur because of edges from a fixpoint variable to the formula it identifies. Alternation freeness now guarantees that not both a ν–variable as well as a μ–variable is reached on a cycle. It is now easy to see that all remaining nodes form trees. Since a formula graph's underlying graph, i.e. the undirected version of the graph, is connected (in the usual sense), 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 simplify the forthcoming algorithm we can define a simplified partition based on the previous lemma (cf. Figure 4).

Theorem 1. *Let φ be a formula that contains at least one fixpoint formula and let $\mathcal{G}(\varphi) = (Q, \rightarrow, \lambda)$ be its graph representation. Then there exists a collection of $Q_1, \dots, Q_m$ such that the following holds:*

1. *The collection of the Q_i is 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 $Q_i \cap Q_j = \emptyset$.*

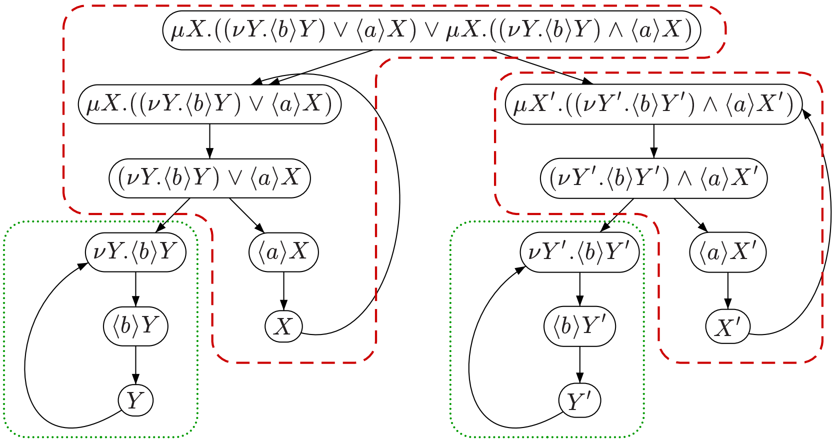


Fig. 4. The partition of the graph for $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$.

2. *Every subgraph induced by Q_i is exactly one of*
   (a) *it contains only μ–cycles (we call the corresponding subgraph a μ–*component*)*
   (b) *it contains only ν–cycles (we call the corresponding subgraph a ν–*component*)*
3. *There exists a tree order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ with an edge from q to q', we have $Q_i \leq Q_j$. More specifically, moves from a configuration in Q_i lead to configurations in either the same Q_i or a child Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$.*

We call the subgraphs of Case 2 (a) and Case 2 (b) μ–components and ν–components, respectively. 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 formulas without any fixpoint formula, we will get a single component which we call arbitrarily a μ–component.

Proof. Let $(Q, \to, \lambda)$ be the graph of φ. By the previous lemma, there exist $Q'_1, \ldots, Q'_{m'}$ that can be partially ordered by $\leq$ and that are either maximal strongly connected components or trees. Let $\lessdot$ denote be the covering relation of $\leq$. The idea is that strongly connected components are united with their children (components) that are trees. Let $i_1, \ldots, i_m \in \{1, \ldots, m\}$ such that $Q'_{i_1}, \ldots, Q'_{i_m}$ comprise exactly the Q'_i that are maximal strongly connected components. For $j \in \{2, \ldots, m\}$ define $Q_j := Q'_{i_j} \cup \bigcup \{Q'_k \mid Q'_{i_j} \lessdot Q'_k \text{ and } Q'_k \text{ is a tree}\}$. Furthermore, if Q'_1 is a tree, then let $Q_1 := Q'_{i_1} \cup \bigcup \{Q'_k \mid Q'_{i_1} \lessdot Q'_k \text{ and } Q'_k \text{ is a tree}\} \cup Q'_1$. Otherwise, Q_1 is defined as usual. It is now a simple matter to show that the Q_i have the desired properties.

Note that for the previous decomposition, it is essential that Lemma 1 provides a tree order on the components. In Section 7.4 we will discuss an alternative definition of a formula's graph which yields partially 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 with respect to the formula's length. Thus, we can label every subformula of a formula by the index of the component within linear time with respect to the length of the formula. We call this number the *component number* of the formula. It is unique by enlarging the partial order of the components to a total order in a deterministic way.

4 Model checking for the μ–calculus

In this section, we recall the notion of model checking and show that this problem for the studied fragment is P–hard, strengthening a similar result by [37] where the propositional μ–calculus is employed. Given a transition system $\mathcal{T}$ and a formula φ, *model checking* is the problem whether $\mathcal{T}$ satisfies φ, i.e. whether $\mathcal{T} \models \varphi$. The *combined complexity* of the model checking problem is its complexity with respect to the product of the size of the transition system and the size of the formula. The *program complexity* of the μ–calculus is the complexity of the model checking problem only with respect to the size of the transition system.

In [37], it was shown that the combined complexity for the alternation free μ–calculus is P–complete and, for a version of the alternation free μ–calculus employing two actions, that its program complexity is P–complete. [22] shows the latter result by using a formula with two propositions. We strengthen both results by employing neither propositions nor any action labelling.

Lemma 2. *The program complexity of the alternation free μ–calculus is P–hard.*

Proof. We reduce the P–complete *Game Problem* [16] to checking a formula of the alternation free μ–calculus with respect to a corresponding labelled transition system. A two player game is a tuple $G = (P_1, P_2, M, W_0, s)$. P_1 and P_2, $P_1 \cap P_2 = \emptyset$, are positions, in which it is the turn of Player 1 or Player 2, respectively. $M \subseteq (P_1 \times P_2) \cup (P_2 \times P_1)$ is the set of moves the respective player can make. $W_0 \subseteq P_1 \cup P_2$ denotes the *succeeding positions*, $s \in P_1$ the *starting position*. The players move alternately beginning with Player 1. We call $x \in P_1 \cup P_2$ *winning* iff either x is succeeding ($x \in W_0$), or $x \in P_1$ and there is a winning $y \in P_2$ such that $(x, y) \in M$, or $x \in P_2$ and for all $(x, y) \in M$, y is winning. The Game Problem is the question whether s is winning.

Corresponding to this, we define a transition system $\mathcal{T}_G = (P_1 \cup P_2, T)$ by $T = (M - \{(p, q) \in M \mid p \in W_0\}) \cup \{(p, p) \in P_1 \times P_1 \mid p \notin W_0$ and there is no transition from p in $M\}$.

T is defined in the way that every deadlock state, i.e. state with no outgoing edges, is a state of W_0 or a state of P_2 in which Player 2 is not able to move. Hence, deadlocks are winning. They can be characterised in the μ–calculus by

$\varphi_{W_0} = [-]\texttt{false}$ where a formula $[-]\varphi$ indicates that φ is satisfied in all successor states. Further winning positions for Player 1 are states of P_1 such that there is a successor state (in P_2) whose direct successors (in P_1) are all winning. Hence, the formula $\varphi = \mu X.(\langle - \rangle [-] X \vee \varphi_{W_0})$ is satisfied in exactly those positions of P_1 which are winning where $\langle - \rangle \varphi$ guarantees the existence of a successor state in which φ holds. Note that φ may be satisfied in further positions of P_2 which does not bother us. We conclude that s is winning in the game $G = (P_1, P_2, M, W_0, s)$ iff $\mathcal{T}_G, s \models \varphi$. Observe that the construction of the transition system can be done in LOGSPACE. Note, we do not make use of propositions. Furthermore, we manage without actions at all by slightly adapting the modal fragment of our logic.

Together with a lineartime algorithm (cf. Section 6 or [9]), we have the following theorem:

Theorem 2 ([5]). *Model checking the alternation free μ–calculus is inherently sequential with respect to the combined complexity as well as the program complexity.*

5 Model checking games for the μ–calculus

In this section, we recall Stirling's characterisation of the model checking problem in terms of games [35]. Games are a simple to understand formalism to deal with model checking the μ–calculus. As we will see, determining the satisfaction of a formula is reduced to colouring 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 a 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 dag–like[5] decomposition. We refer to Section 7.4 for a discussion of the advantages and disadvantages of this approach.

Let in the following $\mathcal{T} = (S, T, \mathcal{A}, s_0, P)$ be a transition system and let φ be a formula over *Var*, *Prop*, and $\mathcal{A}$ with graph $\mathcal{G}(\varphi) = (Q, \rightarrow, \lambda)$. We are now able to define the *model checking game* of $\mathcal{T}$ and φ. Its board is the Cartesian product $S \times Q$ of the set of states and φ's positions. The game is played by two players, namely ∀belard (the pessimist), who wants to show that $\mathcal{T}, s_0 \models \varphi$ does *not* hold, whereas ∃loise (the optimist) wants to show the opposite.

The model checking game $G(s, \varphi)$ 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 ∃loise or ∀belard. We write $\Rightarrow$ instead of $\Rightarrow_{P_i}$ if we abstract from the player. Unlike traditional Ehrenfeucht–Fraïssé games, the players do not have to move alternately. The next turn is not determined by the player but by the current subformula of φ. Hence, the label

[5] directed acyclic graph

of the second component of a configuration C_i determines the player P_i who has to choose the next move. ∀belard makes universal $\Rightarrow_\forall$–moves, ∃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 ∀belard chooses $j = 1$ or $j = 2$, and $C_{i+1} = (s, \pi j)$.
3. $\lambda(\pi) = [K]\psi$, then ∀belard chooses a transition $s \xrightarrow{a} t$ with an action $a \in K$ and $C_{i+1} = (t, \pi 1)$.
4. $\lambda(\pi) = \nu X.\psi$, then $C_{i+1} = (s, \pi 1)$.
5. $\lambda(\pi) = \texttt{true}$, then the play is finished.
6. $\lambda(\pi) = \psi_1 \vee \psi_2$, then ∃loise chooses $j = 1$ or $j = 2$, and $C_{i+1} = (s, \pi j)$.
7. $\lambda(\pi) = \langle K\rangle\psi$, then ∃loise chooses a transition $s \xrightarrow{a} t$ with an action $a \in K$ and $C_{i+1} = (t, \pi 1)$.
8. $\lambda(\pi) = \mu X.\psi$, then $C_{i+1} = (s, \pi 1)$.
9. $\lambda(\pi) = X$ and π' is the position of the fixpoint formula identified by X, then $C_{i+1} = (s, \pi')$.
10. $\lambda(\pi) = p$, then the play is finished.

As the moves 1,4,5, and 8–10 are deterministic, no player needs to be charged with them. With regard to the winning strategies and the algorithm, we will speak of *∀belard–moves* in cases 1–4, 9 and 10 if $\sigma = \mu$, and *∃loise–moves* in all other cases. C_i is called ∀–configuration or ∃–configuration, respectively.

A configuration is called *terminal* if no (further) move is possible. A play $G = C_0 \Rightarrow \cdots$ is called *maximal* iff it ends in a terminal configuration. The *winner* of a maximal play is defined in the following way. If the game is finite, thus ending in a configuration (s, π), then ∀belard wins a play G, if

- $\lambda(\pi) = \texttt{false}$,
- $\lambda(\pi) = \langle K\rangle\psi$,[6] or
- $\lambda(\pi) = p$ and $s \notin P(p)$.

Dually, ∃loise wins a play G, iff

- $\lambda(\pi) = \texttt{true}$,
- $\lambda(\pi) = [K]\psi$, [6] or
- $\lambda(\pi) = p$ and $s \in P(p)$.

An infinite play is won by ∀belard if the outermost fixpoint which is unwinded infinitely often is a μ–fixpoint. Otherwise, the outermost fixpoint which is unwinded infinitely often is a ν–fixpoint and ∃loise wins the game.

Please note that, given a transition system and a formula, there are several plays and these not necessarily have the same winner. 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* now guarantees that

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

the play which P plays according to the rules will be won by P. [35] shows that the model checking problem 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 starting state s, and let φ be a μ–calculus formula. $\mathcal{T}, s \models \varphi$ implies that $\exists$loise has a history–free winning strategy starting at (s, φ), and $\mathcal{T}, s \not\models \varphi$ implies that $\forall$belard has a history–free winning strategy starting at (s, φ). Since a formula either holds or is falsified, this result also implies that model checking games are *determined,* i.e., for every game, either $\forall$belard or $\exists$loise has a winning strategy.

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 $\exists$–configurations and *and*–nodes (denoted by $\bigwedge$) are $\forall$–configurations.

In the following, we concentrate on the alternation free μ–calculus.

The following characterisation of the game graph for this fragment is essential for formulating our sequential and parallel algorithms.

Theorem 3. *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 collection of $Q_1, \ldots, Q_m$ such that the following holds:*

1. *the collection of the Q_i is a partition of Q, i.e., $Q = \bigcup_{i \in \{1, \ldots, m\}} Q_i$ and for all $i, j \in \{1, \ldots, m\}$ with $i \neq j$, it holds $Q_i \cap Q_j = \emptyset$.*
2. *In every subgraph induced by Q_i either μ–formulas and no ν–formulas are unwinded or ν–formulas and no μ–formulas. We call Q_i μ–component or ν–component, respectively.*
3. *There exists a tree order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ with an edge from q to q', we have $Q_i \leq Q_j$. More specifically, moves from a node in Q_i lead to configurations in either the same Q_i or to a child Q_j. Without loss of generality, we may assume that $Q_i \leq Q_j$ implies $i \leq j$.*

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 formulas have component number i. It is a simple task to show that the Q_i have the desired properties.

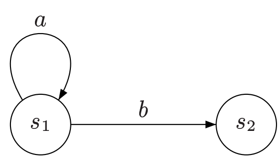


Fig. 5. A transition system

Figure 6 shows a game graph for the transition system in Figure 5 and the formula $\mu X.((\nu Y.\langle b\rangle Y) \vee \langle a\rangle X) \vee \mu X'.((\nu Y'.\langle b\rangle Y') \wedge \langle a\rangle X')$. Note that ∀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. In all forthcoming examples, we write in the second component of a configuration the formula instead of its position to simplify our presentation.

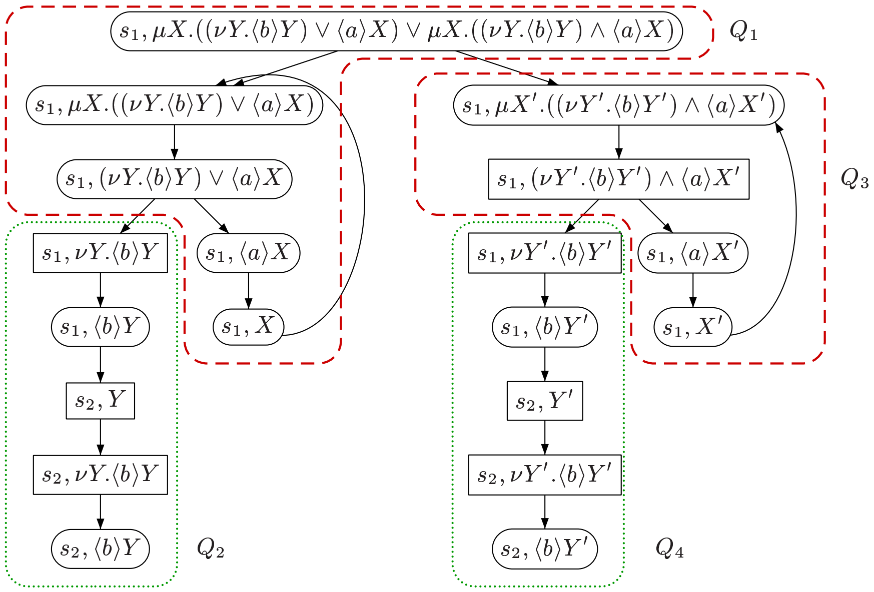


Fig. 6. A game graph and its partition.

Let us fix the decomposition of the game graph shown in the previous proof 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 3 into $Q_1, \ldots, Q_m$ which are tree–like ordered by $\leq$.*
- *Let $Q_{j_1}, \ldots, Q_{j_{k_i}}$ be the components that are children of Q_i. 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:*

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

- *The* initial configurations *of a component are the configurations which are escape configurations for a parent component, and for Q_1, furthermore the initial node of the game graph.*

– *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 $\forall$belard has lost the game. So $\forall$belard tries to leave ν–components. Thus, he tries to move to an escape configuration. Note that the second component of an escape configuration is labelled by a fixpoint formula. Furthermore, $\lfloor Q_i \rfloor = \emptyset$ iff it is a leaf with respect to $\leq$.

The number of components of a game graph's canonical decomposition is identical to the number of the components of the graph of the formula 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 3, it is a constant operation to check the component number of a configuration.

6 Sequential Model Checking

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. For an implementation, though, one would prefer the sequential algorithm presented in [23] since it performs slightly faster in practice and is easier to implement.

The basic idea of both algorithms is labelling a configuration q by *green* or *red*, depending on whether $\exists$loise or $\forall$belard has a winning strategy for the game starting in this configuration q. Furthermore, they both employ the canonical decomposition of the game graph according to Definition 2. They differ in the order the several components are processed. The first algorithm proceeds bottom–up, the second top–down.

Figure 6 shows a simple game graph and its canonical decomposition into disjoint sets Q_i that are μ– or ν–components (cf. Definition 2). Furthermore, for every $q \in Q_i$ and $q' \in Q_j$ for which there is a possible move from q to q', we have that $i \leq j$. Thus, every infinite play gets trapped within a single Q_i, and the winner depends on the fact whether a μ–formula or a ν–formula is unwinded in Q_i. For the example shown in Figure 6, we obtain four components, $Q_1, \ldots Q_4$.

6.1 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 escape configurations and assume that every configuration in $\lfloor Q_i \rfloor$ is either labelled by *green* or *red* expressing that either $\exists$loise or $\forall$belard has a winning strategy from this configuration, respectively. It is now obvious, that every play starting in 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`, a box formula, or a proposition which is valid in the current state then the configuration is coloured with *green*. Otherwise, the formula component is `false`, a diamond formula, or a non–satisfied proposition and the configuration is coloured with *red*.

Once a configuration $q \in Q_i \cup \lfloor Q_i \rfloor$ is labelled by *red* or *green*, its predecessors are labelled if possible. That means, an $\bigwedge$–node q' is labelled by *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. A simple but useful insight is the following.

Lemma 3. *Once a configuration obtained a colour, the colour is never changed.*

Corollary 1. *The colouring process is terminating.*

However, the labelling process may leave some configurations of the component Q_i uncoloured. Let us now understand, that all remaining uncoloured configurations can be labelled by *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 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 then 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. 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 6. We have four components, $Q_1, \ldots Q_4$. 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 by *red*. Now, propagating the colours to the predecessor configurations will colour every configuration of Q_2 with *red*.

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

The next component to handle with respect to 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*.

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 are propagated within Q_1. The current situation is depicted in Figure 7 in which *red* configurations are filled with ■. Now, the second phase of colouring a component comes into play. All remaining configurations will be labelled by *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) with respect to the size of the game graph. The size of the game graph is bounded by the size of the underlying transition system times the length of the formula, i.e. it is bounded by $s \times l$ where s is the size of the underlying transition system and l is the length of the formula. 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*.

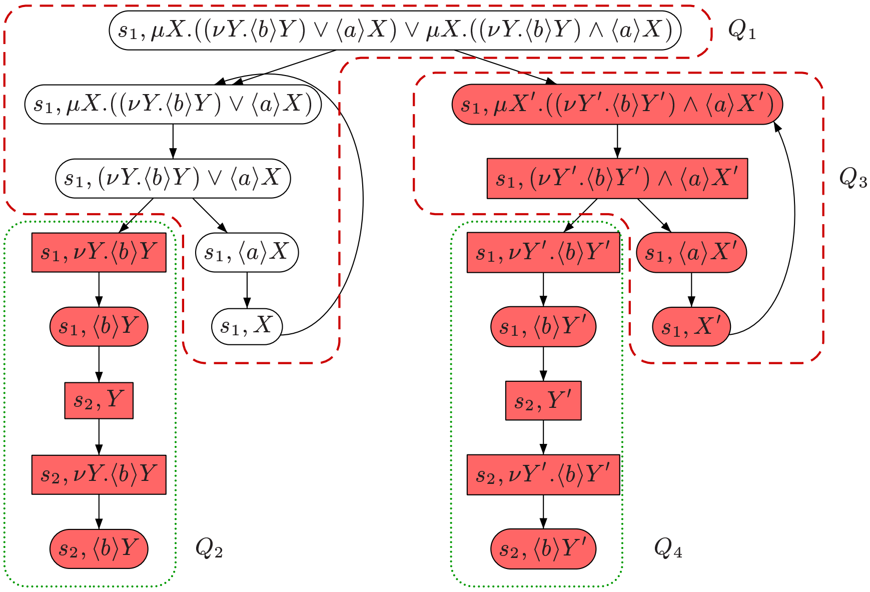


Fig. 7. The situation before the second phase

6.2 Colouring top–down

For the second algorithm, let us assume that the game graph is again partitioned into components Q_i which form a tree. To get a result with considering 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. Anyway, 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 this configuration. That means, an $\bigwedge$–node is labelled by *red* if one successor is *red*, labelled by *lightred*, if no successor is *red* but at least one is *lightred*, labelled *lightgreen*, if all successors are *lightgreen* or *green* (and one is indeed *lightgreen*), and labelled by *green*, if all successors are *green*. In all other cases, the configuration remains unlabelled. An $\bigvee$–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 the following result.

Lemma 4. *Once a configuration obtained a full colour, this is never changed. A light colour is only changed to the corresponding full colour.*

Corollary 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 by *red*.

Theorem 5. *For any game starting in a configuration without a colour,* $\forall$*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. $\forall$belard's strategy will be any choosing one uncoloured successor. Then he will win every play. Every uncoloured $\exists$loise–configuration has *red*, or uncoloured successors, so $\exists$loise has the choice to move to a configuration which is winning for $\forall$belard or to move to an uncoloured configuration. $\forall$belard will choose in an uncoloured configuration an uncoloured successor, or, if $\exists$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 $\forall$belard has a winning strategy, or will stay infinitely often within Q_i and $\forall$belard wins.

Again, the previous theorem is crucial and allows a powerful parallel version of this algorithm.

Our component now may contain configurations which are coloured with *lightgreen*. However, we cannot guarantee that $\exists$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 6. 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 by *lightgreen* (cf. Figure 8[7]).

Propagating the colour information will colour every configuration of Q_1 with *lightgreen* which is shown in Figure 9.

[7] *lightgreen* configurations are identified by writing their label in *lightgreen*.

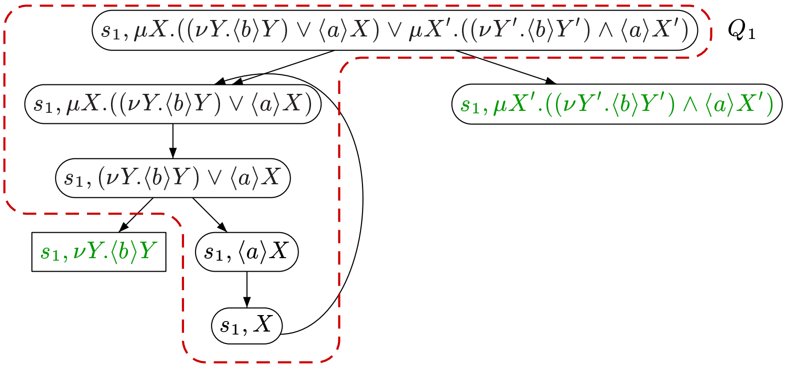


Fig. 8. Starting with Q_1.

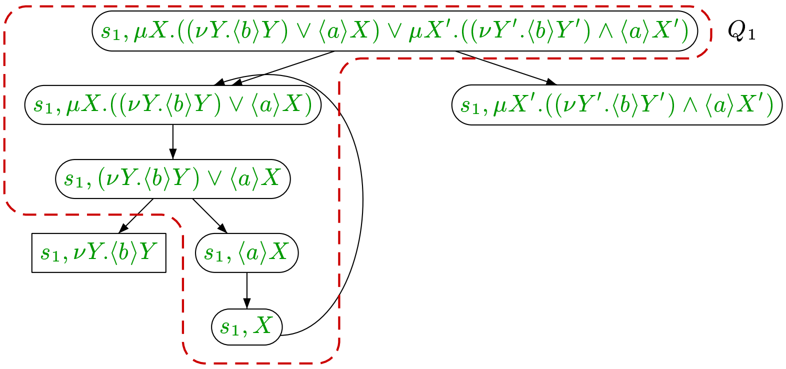


Fig. 9. Propagating assumptions.

The subsequent phase of colouring *white*–configurations *red* and *lightgreen*–configurations *white* will turn the whole component to a complete *white* one so that it 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 to 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 11 shows the coloured game graph right before recolouring the speculatively *lightgreen* coloured configurations back to *white*.

We turn our attention now 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*. However, the propagation does not influence the preceeding $\bigwedge$–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.

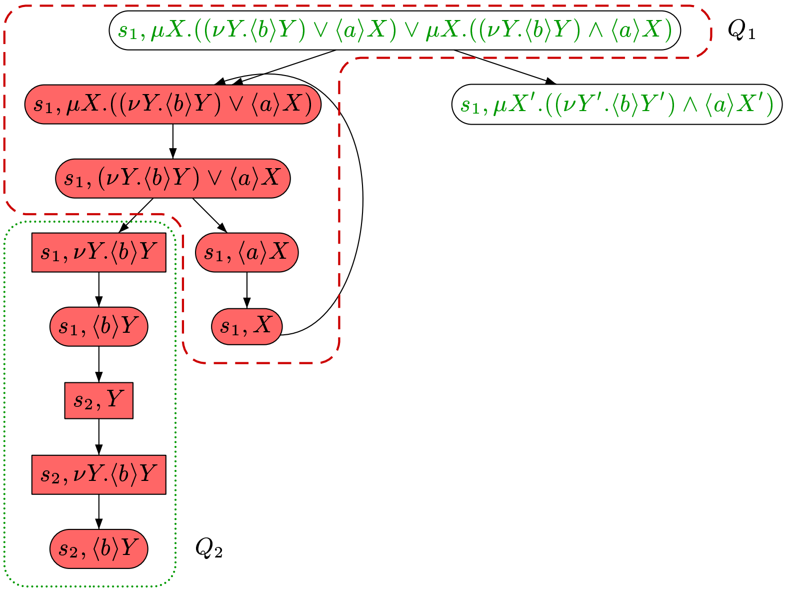


Fig. 10. Propagating assumptions for Q_2 and drawing conclusions.

Figure 11 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 fixpoint formulas 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.

7 Parallel Model Checking

Given a transition system and an L_μ^1–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 to handle each component in parallel but to handle to components sequentially in a top–down or bottom–up manner.

7.1 Distributing the game graph

Let us first discuss how to construct (a component) of the game graph in parallel. It is obvious that this can be carried out by a typical breadth–first strategy.

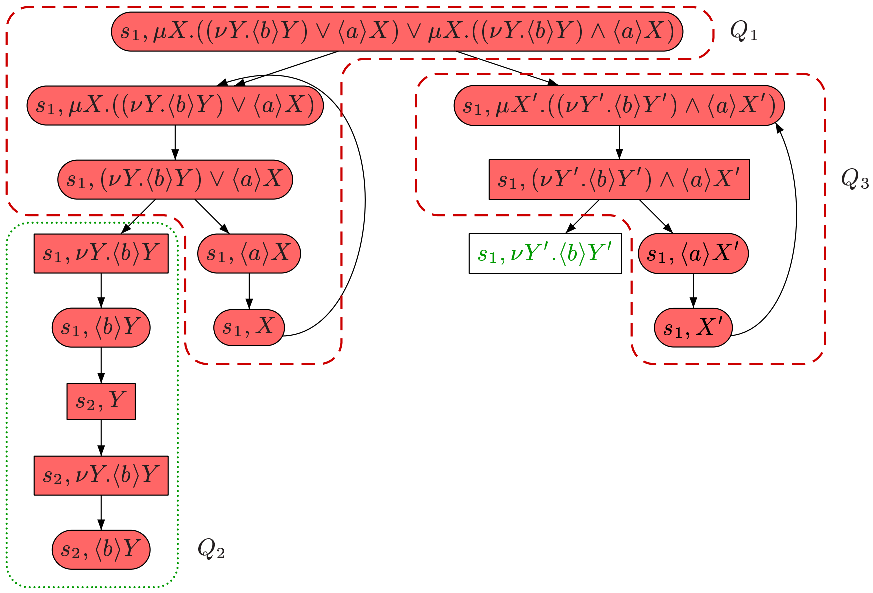


Fig. 11. Propagating assumptions for Q_3 and drawing conclusions.

Given a node q, determine its successors $q_1, \ldots, q_n$. Now, the successors can be processed in the same manner in parallel. However, to obtain a terminating procedure, only exactly the q_i *not* processed before must be expanded. All states generated must be stored within the NOW, and load sharing must be guaranteed. On a shared memory architecture, this does not involve big conceptual problems. For distributed memory machines, however, this is a little bit more difficult.

A first idea might be to distribute the first $q_1, \ldots, q_n$ to the first n processors, and these process the q_i as described before and distribute the successors to the next processors. However, deciding whether a q_i was processed before becomes an expensive operation. Every processor could have processed q_i and should therefore be consulted. In the worst case, for every node, such a broadcast is required. This yields no reasonable algorithm.

A different, often–employed way to store graphs on a distributed memory machine is to divide the graph's adjacency matrix $M \in \{0,1\}^{|Q|\times|Q|}$ into equal sized *blocks* and to store each block on a single processor [33]. This has several advantages. First, the blocks of the matrix can be generated in parallel. Second, given nodes $p, q \in V$, it is easy to check whether there is an edge from p to q, i.e. whether $M_{p,q} = 1$. Since there is a unique location for the block of the matrix containing the value for the pair (p, q), a single communication is needed. Third, every processor gets the same amount of data. This promises a similar load for every processor. Furthermore, we want to mention that several typical graph algorithms can be formulated in terms of matrix operations and admit a good parallelisation [33].

However, for our problem this approach cannot be applied. First, our resulting adjacency matrix would be sparse and storing it completely would yield too much overhead. Storing only the non–zero entries, on the other hand, does not guarantee an equal load for the processors. But more importantly, the number of nodes of our graph is not known a priori but computed while constructing the graph.[8] Hence, the partition of the whole game graph into blocks cannot be determined in advance.

We propose the following way to construct and store the graph which is inspired by the work pool presentation of [25] and is similarly applied in [34]. First, we employ adjacency lists instead of matrices. Note that we need also links to the predecessors as well as to the successors of a node for the labelling algorithm. Let f be a function mapping the states of the game graph to a processor of our network. Usually, one takes a function in the spirit of a hash function assigning to every state an integer and subsequently its value modulo the number of processors. Then, f determines the location of every state within the network uniquely and without global knowledge. In a breadth–first manner, starting with the initial state q_0 of the game graph, the state space can be constructed in parallel with the help of f in the following way. Given a state q (and possibly some of its direct predecessors), send it to its processor $f(q)$. 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 new computed successors $q_1, \ldots, q_k$. These are sent in the same manner to their (with respect to f) processors, together with the information that q is a direct predecessor. The corresponding processes update their local memory similarly.

In terms of the matrix implementation, for every q_i a unique processor $f(q_i)$ is determined on which the non–zero entries of the row q_i and column q_i of M are stored. Checking whether a q_i was processed before can be checked locally on $f(q_i)$. Accessing the successors and predecessors of a node is as simple as in the matrix implementation. The load for the processors depends on the function f. Hence, we have similar advantages as in the case of the matrix implementation while avoiding that the maximal number of nodes has to be known in advance.

7.2 Labelling the game graph

Given the game graph, a first possibility for labelling the nodes with *red* and *green* would be to apply a depth–first–search as done by the sequential algorithm described in [23]. However, it is not clear how to do this in parallel efficiently.

Another possibility for labelling the nodes with *red* and *green* is applying typical algorithms for generating strongly connected components of our game graph in parallel which will be labelled in a second step [33]. However, labelling

[8] In the context of model checking, the transition system is not given explicitly but expanded at run–time from a formal system description.

a connected component requires the knowledge whether the component can be "successfully abandoned" (cf. Section 5). It is neither clear how to obtain this information from the graph computed by these algorithms nor to modify them in the way that this information can be gathered easily.

We propose to extend the sequential algorithms described Section 6 towards a parallel implementation. As explained in the previous subsection, 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 6. 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 6). It is no problem to do this recolouring on the distributed game graph in parallel. Thanks to Theorem 4 or Theorem 5, 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 the DFG token termination algorithm as presented in [10]. It has the advantage that it scales well with respect to 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 6. Thus, we get a set of algorithms differing in the order the components are processed.

Theorem 6. *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 by red.*

7.3 The algorithm

To describe our approach in more detail, we show several fragments of our algorithm in *pseudo code*. Especially, we show that the two steps of constructing the game graph and labelling the nodes can carried out in an integrated way.

We assume that the game graph is represented as an array with a dimension according to the number of the components. Every array element is a record of two items. The first is a list (or, more efficiently tree) of nodes. Each node is a record containing the label of the current configuration, its colour, and a list of (pointers to) predecessors and successors.[9] The second item of an array element is a list of pointers to the initial nodes of a component.

The most important function is shown in Figure 12. 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 correct (full) colour, a colouring process is initiated, if a terminal configuration is reached.

[9] For configurations owned by the current workstation (according to f), we use pointers, of course.

```
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(comp)
          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
```

Fig. 12. Expanding a component

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 processing 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 a colour, if the current node has one. Otherwise, the node is added to the right component of the graph (21). The configuration and the predecessor are stored (22,23). A colouring process is initiated, if the current node is a terminal one (24–28). Furthermore, if the node is neither terminal or an escape configuration, it successors are are considered (30). Thus, its successors computed and expanded (2–4). Furthermore, the successors together with an initially *white* colour are stored in the current node (5–6).

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 13) can be called in a bottom up manner.[10] 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.

```
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
```

Fig. 13. The recolour function

For the bottom–up algorithm, the colour can be computed as described in Figure 14.

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. After that, a recolour process (Figure 13) is started.

There are several possibilities to process the components. In the examples shown in Section 6, 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

[10] Don't bother about the *light* colours which are only used in the top–down version.

```
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
```

Fig. 14. The compute colour function

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.

7.4 Variations of the algorithm and optimisation issues

We already mentioned in the previous section that there are several possibilities to process the components. Note that only the colours of the escape configurations are needed when colouring a component. Thus, all other nodes of a child component can be deleted for colouring the current component. Furthermore, a coloured configuration is of no use if all its predecessors are coloured (with a full colour). Especially in the top–down algorithm, this may be used to reduce the size of a component before the child components are considered.

In general, it is possible to design the algorithm in the way that only a single component is totally stored in the workstation cluster while from the other components only the initial/escape configurations are needed. The price to pay for this implementation is that some components have to be reconstructed.

Another variant of our algorithm can be obtained by taking subformulas instead of the occurrence set of the subformulas 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. It can be expected, that the resulting game graph is smaller since nodes may be shared. However, the tree order simplifies the decision when a component can be removed, as described in the previous paragraph.

7.5 Winning strategies

As pointed out already when motivating the framework 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 for 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 proofs of Theorem 4 and Theorem 5 that the right strategy is choosing a previously *white* successor.

8 Experimental Results

We have tested our approach within our verification platform TRUTH [27] implemented in Haskell as well as with a separated stand–alone version written in C++. In both versions, we have tested a depth–first top–down approach (without using any optimisations as for example the one mentioned in Section 7.4), so far.

8.1 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[11], which enabled us to embed this algorithm in the infrastructure of our verification tool TRUTH and also to prototype a *concise* reference implementation. The actual Haskell source code of the algorithm has less than 300 lines of code. The communication layer of our implementation is based upon MPICH[12], an implementation of the MPI (Message Passing Interface) standard.

Figure 15 and Figure 16 show the measured results of state distribution and the speedup when running our implementation on a NOW consisting of up to 52 processors and a total of 13GB main memory. They are connected with a conventional 100MBit Fast–Ethernet network.

The distribution routine shows that our approach is very well suited for constructing large game graphs. We were able to construct graphs with several hundred thousands of states within minutes. The game graph of the largest example we have constructed so far with this version, a quad–parallel instance of the Alternating Bit Protocol [29], consists of more than 1.6 million states, and we get a homogeneous distribution of the state space on the workstations (Figure 15). The distribution depends on our hash function f, and the results are quite good compared to its simplicity.[13] In fact, all our measurements showed similar re-

[11] http://haskell.org/

[12] http://www-unix.mcs.anl.gov/mpi/mpich/

[13] Since our distribution function is based on the underlying CCS terms supported in TRUTH we do not present the function in this report.

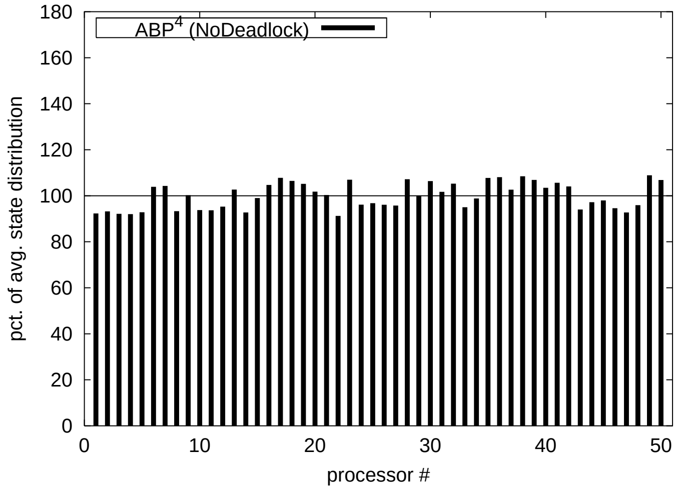


Fig. 15. State distribution

sults, provided the size of the graph is reasonably larger than the number of used workstations.

Our approach also scales very well with regard to the overall runtime (Figure 16). 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.

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.

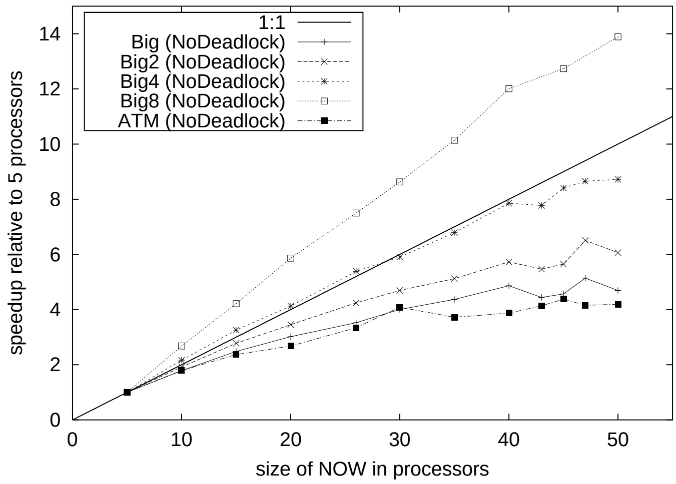


Fig. 16. Runtime results

8.2 The C++ version

We have implemented a simple prototype tailored for transition systems computed by the μCRL tool set [4]. We have tried the version—for example—on a transition system with 1 million of states. We were able to check a mutual exclusion property within 9 minutes on the NOW. The overall game graph had 13 millions of states. 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 (cf. Figure 17).

The reason is that in contrast to the Haskell version where the transition system is computed on–fly–out from a given CCS Process equation system, the system is already computed in the C++ version. 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.

Anyway, our C++ 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.

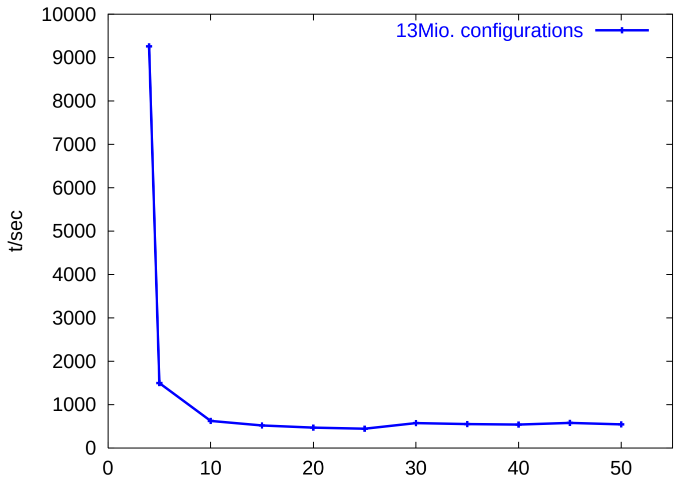


Fig. 17. Run time results (C++)

9 Conclusion

In this paper, we 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 our approach within the verification platform TRUTH. We found out that the algorithm scales very well with respect to run–time and memory consumption when enlarging the NOW. Furthermore, the distribution of states on the processors is homogeneous.

While our approach is already of practical interest since it allows to check larger systems, there is still a lot of work to be done in the area of parallel model checking. Different (especially non–P–complete) specification logics might provide better parallel model checking algorithms. Furthermore, it is a natural question how to extend our approach towards *fair* model checking [7].

However, 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.

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. G. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 304–312, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.
4. 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, *Proceedings of the 13th Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag Inc., July 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, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 543–558. Springer-Verlag Inc., Apr. 2001.
6. G. Cabodi, P. Camurati, and S. Que. Improved reachability analysis of large FSM. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 354–360. IEEE Computer Society Press, June 1996.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
8. E. M. Clarke and J. Wing. Formal methods: State of the art and future directions. Technical Report CMU-CS-96-178, Carnegie Mellon University (CMU), Sept. 1996.
9. 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 *LNCS*, pages 48–58, Berlin, Germany, July 1992. Springer.
10. 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.
11. E. A. Emerson. *Model checking and the mu-calculus*, volume 31 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, chapter 6. American Mathematical Society, 1997.
12. 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.
13. T. M. P. I. Forum. Document for a Standard Message-Passing Interface. CS-93-214, University of Tennessee, 11 1993.
14. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Pres, 94.
15. S. Gnesi, D. Latella, G. Lenzini, C. Abbaneo, A. Amendola, and P. Marmo. A formal specification and validation of a critical system in presence of byzantine errors. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, number 1785 in LNCS. Springer, 2000.
16. R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.
17. O. Grumberg, T. Heyman, and A. Schuster. Distributed symbolic model checking for μ-calculus. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 350–362. Springer-Verlag Inc., July 2001, to appear.
18. T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In O. Grumberg, editor, *Computer Aided Verification, 12th International Conference*, volume 1855 of *Lecture Notes in Computer Science*, pages 20–35. Springer-Verlag, June 2000.

19. A. J. Hu, G. York, and D. L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *31st Design Automation Conference*, pages 276–282, 1994.
20. D. Janin and I. Walukiewicz. On the expressive completeness of the propositional mu-calculus with respect to monadic second order logic. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 263–277, Pisa, Italy, 26–29 Aug. 1996. Springer-Verlag.
21. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, Dec. 1983.
22. 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.
23. M. Lange. Spielbasiertes Model-Checking für den alternierungsfreien mu-Kalkül. Master's thesis, Aachen, University of Technology, 1999. (German).
24. M. Lange, M. Leucker, T. Noll, and S. Tobies. Truth – a verification platform for concurrent systems. In *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science. Springer-Verlag Wien New York, 1999.
25. B. P. Lester. *The Art of Parallel Programming*. Prentice Hall, 1993.
26. 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*, volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 77–91. Springer, 1999.
27. M. Leucker and T. Noll. Truth/SLC - A parallel verification platform for concurrent systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 255–259. Springer-Verlag Inc., July 2001.
28. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.
29. R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.
30. A. A. Narayan, J. J. J. Isles, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Reachability analysis using partitioned–roBBDs. In *Proceedings of the IEEE International Conference on Computer Aided Design*, pages 388–393. IEEE Computer Society Press, June 1997.
31. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, New York, 1994.
32. D. Peled. Ten years of partial order reduction. In *CAV, Computer Aided Verification*, number 1427 in LNCS, pages 17–28, Vancouver, BC, Canada, 1998. Springer.
33. S. H. Roosta. *Parallel Processing and Parallel Algorithms*. Springer, New York, Berlin, Heidelberg, 1999.
34. U. Stern and D. L. Dill. Parallelizing the Murφ verifier. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer-Verlag, June 1997. Haifa, Israel, June 22-25.
35. C. Stirling. Games for bisimulation and model checking, July 1996. Notes for Mathfit Workshop on finite model theory, University of Wales, Swansea,.
36. A. L. Stornetta. Implementation of an efficient parallel BDD package. Master's thesis, University of California, Santa Barbara, 1995.
37. S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 154–163, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

97-01	Jahresbericht 1996
97-02	J. Faassen: Using full parallel Boltzmann Machines for Optimization
97-03	A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems
97-04	M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler
97-05 *	S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge
97-06	M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries
97-07	P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen
97-08	D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting
97-09	C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets
97-10	M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases
97-13	M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs
97-14	R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System
97-15	G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms
98-01 *	Jahresbericht 1997
98-02	S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes
98-03	S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr
98-04 *	O. Kubitz: Mobile Robots in Dynamic Environments
98-05	M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems
98-07	M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
98-08 *	H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen
98-09 *	Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 * Jahresbericht 1998

99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge / Marcin Jurdziński: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 * Mareike Schoop: Cooperative Document Management

2000-06 * Mareike Schoop, Christoph Quix (Ed.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.