Parallel Model Checking for LTL, CTL*, and L^2_μ

Martin Leucker [1] Rafał Somla [2]

IT Department
Uppsala University
Uppsala, Sweden

Michael Weber [3]

Lehrstuhl für Informatik II
RWTH Aachen
Aachen, Germany

Abstract

We describe a parallel model-checking algorithm for the fragment of the μ-calculus that allows one alternation of minimal and maximal fixed-point operators. This fragment is also known as L^2_μ. Since LTL and CTL* can be encoded in this fragment, we obtain parallel model checking algorithms for practically important temporal logics.

Our solution is based on a characterization of this problem in terms of two-player games. We exhibit the structure of their *game graphs* and show that we can iteratively work with game graphs that have the same special structure as the ones obtained for L^1_μ-formulae. Since good parallel algorithms for colouring game-graphs for L^1_μ-formulae exist, it is straightforward to implement this algorithm in parallel and good run-time results can be expected.

1 Introduction

Model checking [6] is a key tool for verifying complex hardware and software systems. However, the so-called *state-space explosion* still limits its application. While *partial-order reduction* or *symbolic model checking* reduce the

[1] Email: Martin.Leucker@it.uu.se This author is supported by the European Research Training Network "Games".
[2] Email: Rafal.Somla@it.uu.se
[3] Email: michaelw@informatik.rwth-aachen.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

state space by orders of magnitude, typical verification tasks still take modern sequential computers to their memory limits. One direction to enhance the applicability of today's model checkers is to use the accumulated memory (and computation power) of parallel computers. Thus, we are in need of new parallel model checking algorithms.

A well-known logic for expressing specifications is Kozen's μ-calculus [10], a temporal logic offering Boolean combinations of formulae and, especially, labelled *next*-state, minimal and maximal fixed-point quantifiers. The (dependent) nesting of minimal- and maximal fixed-point operators forms the alternation depth hierarchy of the μ-calculus. It is well known that it is strict [4]. The complexity of model checking, on the other hand, grows exponentially with the alternation depth for all known algorithms. It is then reasonable to limit the alternation depth of a formula to a practical important level and to develop efficient algorithms for this class of problems. In particular, we only need alternation depth 2 to capture the expressive power of LTL and CTL* [7].

In this paper we develop a parallel model checking algorithm for μ-calculus formulae up-to alternation depth 2. This fragment is known as L^2_μ. Our algorithm uses a characterization of the model-checking problem for this fragment in terms of two-player games [8,12]. Strictly speaking, we present a parallel algorithm for colouring a game graph in order to answer the underlying model-checking problem. We show that the game graph can be decomposed into components that can (after some simple modifications) easily be coloured using a colouring algorithm for games obtained by formulae of L^1_μ. L^1_μ is the alternation-free fragment of the μ-calculus, and hence $L^1_\mu \subset L^2_\mu$.

In [3], a parallel colouring algorithm for game graphs based on L^1_μ was introduced. We show that this algorithm is helpful as a subroutine in colouring graphs for L^2_μ. Thus, our reduction allows a simple and promising approach to check formulae of LTL, CTL*, and L^2_μ.

The area of parallel model checking has gained interest in recent years. In [11], a parallel reachability analysis is carried out. The distribution of the underlying structure is similar to the one presented here but their algorithm is not suitable for model checking temporal-logic formulae. A notable step is done in [9], in which a symbolic parallel algorithm for the full μ-calculus is introduced. [5] presents a model-checking algorithm for LTL using a costly parallel cycle detection. Another model-checking algorithm for LTL are introduced in [1], based on a nested depth-first search. Our approach, however, explicitly uses the structure of game graphs for L^2_μ.

In Section 2, we fix some notions on graphs, recall the syntax and semantics of the μ-calculus, the definition of model checking, and describe model-checking games for the μ-calculus. We analyze fixed-point computations and the structure of game graphs in Section 3. We elaborate on a sequential model checking algorithm in this section as well. The corresponding parallel model-checking procedure is shown in Section 4.

2 Graphs, μ-Calculus, and Games

2.1 Graphs

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

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

2.2 The μ-Calculus

Let *Var* be a set of fixed-point variables and $\mathcal{A}$ a set of actions. Formulae of the modal μ-calculus over *Var* and $\mathcal{A}$ in positive form as introduced by [10] are defined as follows:

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

where $X \in \mathit{Var}$ and $K \subseteq \mathcal{A}$.[4] For a formula φ of the μ-calculus, we introduce the notion of *subformulae* (denoted by $\mathit{Sub}(\varphi)$), *free* and *bound* variables, and *sentences* as usual.

A formula φ is *normal* iff every occurrence of a binder μX or νX in φ binds a distinct variable. For example, $(\mu X.X) \vee (\mu X.X)$ is not normal but $(\mu X.X) \vee (\mu Y.Y)$ is. By renaming, any 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 Φ_X such that $\sigma X.\Phi_X \in \mathit{Sub}(\varphi)$. We call X a ν-variable iff it is bound by the ν binder and μ-variable otherwise. We call φ a μ-formula iff $\varphi = \mu X.\Phi_X$ for appropriate X. ν-formulae are introduced analogously. For $\varphi = \varphi_1 \wedge \varphi_2$, $\varphi = \varphi_1 \vee \varphi_2$, $\varphi = [K]\varphi_1$, $\varphi = \langle K \rangle \varphi_1$, $\varphi = \nu X.\varphi_1$, or $\varphi = \mu X.\varphi_1$, we call φ_1 and φ_2 a direct subformula of φ. From now on, we assume all formulae to be normal.

A μ-calculus formula is interpreted over a *labelled transition system* $\mathcal{T} = (S, T, \mathcal{A}, s_0)$ where S is a finite set of states, $\mathcal{A}$ a set of actions, and $T \subseteq S \times \mathcal{A} \times S$ denotes the transitions. As usual, we write $s \xrightarrow{a} t$ instead of $(s, a, t) \in T$ and $s \not\xrightarrow{a}$ if there exists no $t \in S$ with $s \xrightarrow{a} t$. Furthermore, let $s_0 \in S$ be the *initial state* of the transition system.

Given a transition system $\mathcal{T}$ and a variable valuation $V : \mathit{Var} \rightarrow 2^S$ the semantics of a formula φ over *Var* is a set of states $[\![\varphi]\!]_V \subseteq S$ defined as follows

- $[\![\texttt{true}]\!]_V = S, \quad [\![\texttt{false}]\!]_V = \varnothing, \quad [\![X]\!]_V = V(X),$

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

- $[\![\varphi_1 \wedge \varphi_2]\!]_V = [\![\varphi_1]\!]_V \cap [\![\varphi_2]\!]_V$, $[\![\varphi_1 \vee \varphi_2]\!]_V = [\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V$,
- $[\![[K]\varphi]\!]_V = \{s \mid \forall_{s \xrightarrow{a} s'}\, a \in K \Rightarrow s' \in [\![\varphi]\!]_V\}$,
- $[\![\langle K\rangle\varphi]\!]_V = \{s \mid \exists_{s \xrightarrow{a} s'}\, a \in K \wedge s' \in [\![\varphi]\!]_V\}$,
- $[\![\mu X.\Phi_X]\!]_V$ is the minimal fixed point of the monotonic function $F_X^V : 2^S \to 2^S$ given by $F_X^V(A) = [\![\Phi_X]\!]_{V[X \leftarrow A]}$,
- $[\![\nu X.\Phi_X]\!]_V$ is the maximal fixed point of F_X^V.

Note that if φ is a sentence then the set $[\![\varphi]\!]_V$ does not depend on V and in that case we can write $[\![\varphi]\!]$ to denote the semantics of φ. We write $\mathcal{T}, s \models \varphi$ and say that sentence φ is satisfied in state s of a transition system $\mathcal{T}$ if $s \in [\![\varphi]\!]$.

The fixed points can be computed by the standard iteration method. The domain of the function is the finite complete partial order (cpo) 2^S where each chain has length at most $|S|$. Hence, to find the fixed point we need at most $|S|$ iterations.

We say that variable X *subsumes* variable Y (notation $Y \sqsubseteq X$) if $\sigma Y.\Phi_Y$ is a subformula of Φ_X. Note that if X subsumes Y then each computation of $[\![\Phi_X]\!]_V$ potentially leads to a nested fixed-point computation of $[\![\sigma Y.\Phi_Y]\!]_{V[X \leftarrow A]}$. If X does not occur in Φ_Y then the nested fixed point needs to be computed only once since its value does not depend on the current value of X. Otherwise the nested fixed point must be recalculated each time the current value of X is updated. We say that variable Y *depends* on X (notation $Y \prec X$) if $Y \sqsubseteq X$ and X occurs free in Φ_Y.

Definition 2.1 [alternation depth] The alternation depth $\mathrm{ad}(\varphi)$ of a formula φ is the length of the longest chain $X_1 \prec X_2 \prec \cdots \prec X_d$ of variables occurring in φ such that X_i and X_{i+1} are of different type for each i.

Alternation depth can be also defined for a single variable. In that case we look at chains of alternating variables which end up in the given one. Formally, the alternation depth $\mathrm{ad}(X)$ of a variable $X \in \mathit{Var}(\varphi)$ is the length of the longest chain $X_1 \prec X_2 \prec \cdots \prec X_d = X$ of variables of alternating types. Clearly $\mathrm{ad}(\varphi) = \max\{\mathrm{ad}(X) \mid X \in \mathit{Var}(\varphi)\}$.

We denote by L_μ^n the set of all μ-formulae up to alternation depth n and by L_μ the set of all μ-calculus formulae.

2.3 Model-checking games for the μ-calculus

Let us recall Stirling's definition of model checking games [12].

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

The model checking game $G(s, \varphi)$ for a state s and a formula φ is given by all its *plays*, i.e. (possibly infinite) sequences $C_0 \Rightarrow_{P_0} C_1 \Rightarrow_{P_1} C_2 \Rightarrow_{P_2} \ldots$

of *configurations*, where for all i, $C_i \in S \times \mathrm{Sub}(\varphi)$, $C_0 = (s, \varphi)$, and P_i is either ∃loise or ∀belard. We write $\Rightarrow$ instead of $\Rightarrow_{P_i}$ if we abstract from the players. Each next turn is determined by the the second component of current configuration, i.e. the subformula of φ. ∀belard makes universal $\Rightarrow_\forall$-moves, ∃loise makes existential $\Rightarrow_\exists$-moves. More precisely, whenever C_i is

(i) $(s, \texttt{false})$, then the play is finished.

(ii) $(s, \psi_1 \wedge \psi_2)$, then ∀belard chooses $\psi' = \psi_1$ or $\psi' = \psi_2$, and $C_{i+1} = (s, \psi')$.

(iii) $(s, [K]\psi)$, then ∀belard chooses $s \xrightarrow{a} t$ with $a \in K$ and $C_{i+1} = (t, \psi)$.

(iv) $(s, \nu X.\psi)$, then $C_{i+1} = (s, \psi)$.

(v) $(s, \texttt{true})$, then the play is finished.

(vi) $(s, \psi_1 \vee \psi_2)$, then ∃loise chooses $\psi' = \psi_1$ or $\psi' = \psi_2$, and $C_{i+1} = (s, \psi')$.

(vii) $(s, \langle K\rangle\psi)$, then ∃loise chooses $s \xrightarrow{a} t$ with $a \in K$ and $C_{i+1} = (t, \psi)$.

(viii) $(s, \mu X.\psi)$, then $C_{i+1} = (s, \psi)$.

(ix) (s, X), then $C_{i+1} = (s, \sigma_X.\Phi_X)$.

We will speak of ∀*belard-moves* in cases (i)–(iv) and (ix) if $\sigma = \mu$, and ∃*loise-moves* in all other cases. C_i is called ∀-configuration or ∃-configuration, respectively. A move according to (ix) is also called an *unwinding* of X.

A configuration is called *terminal* if no (further) move is possible. A play G is called *maximal* iff it is infinite or ends in a terminal configuration. The *winner* of a maximal play is defined in the following way: If the play is finite, ending in a configuration (s, ψ), then ∀belard wins G iff $\psi = \texttt{false}$ or $\psi = \langle K\rangle\psi$.[5] Dually, ∃loise wins G iff $\psi = \texttt{true}$ or $\psi = [K]\psi$.[5] An infinite play is won by ∀belard iff the outermost fixed point that is unwinded infinitely often is a μ-fixed point. Otherwise, when the outermost fixed point that is unwinded infinitely often is a ν-fixed point, then ∃loise wins the game.

A *strategy* is a set of rules for a player P telling her or him how to move in the current configuration. It is called *history free* if the strategy only depends on the current configuration without considering the previous moves. A *winning strategy* guarantees that the play that P plays according to the rules will be won by P. [12] shows that model checking for the μ-calculus is equivalent to finding a history-free winning strategy for one of the players: Let $\mathcal{T}$ be a transition system with state s and φ a μ-calculus formula. $\mathcal{T}, s \models \varphi$ implies that ∃loise has a history-free winning strategy starting in (s, φ), and $\mathcal{T}, s \not\models \varphi$ implies that ∀belard has a history-free winning strategy starting in (s, φ).

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

The existence of a history-free winning strategy implies that every node

[5] Note that, due to maximality, we have $\nexists t \ : \ s \xrightarrow{a} t$ for any $a \in K$.

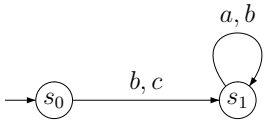


Fig. 1. A transition system.

of the game graph can be labelled with a colour identifying which player has a winning strategy for every game starting from this node. We use *red* for ∀belard and *green* for ∃loise. Determining a winning strategy can therefore be understood as a colouring problem.

3 Sequential Model Checking

In this section we explain how model checking for L^2_μ-formulae can be carried out by iteratively using a procedure for checking L^1_μ-formulae. Firstly, we describe how to reduce alternation depth in the usual fixed point computation. In the next subsection we show how this observation has to be read when dealing with game graphs. As a result, we get a game-based sequential model checking algorithm for L^2_μ-formulae, which employs a game-based algorithm for L^1_μ-formulae. In the next section, we describe in which way this algorithm can be parallelized.

3.1 Reducing alternation depth.

The basic idea for reducing alternation depth is to resolve fixed point variables of the maximal alternation depth using the iteration method. In the remaining computations only formulae with lower alternation depth are involved.

As a running example, let us study the fixed point computation for the transition system shown in Figure 1 and the formula $\mu X.[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X)$. The formula contains two variables X and Y of alternation depth 2 and 1 respectively. We compute the value of X in an iteration starting from $X = \varnothing$. The next approximation is given by

$$[\![\Phi_X]\!]_{V[X\leftarrow\varnothing]} = \{s \mid s \overset{c}{\not\rightarrow}\} \cap [\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\varnothing]}$$

The inner fixed point formula (with value of X fixed to $\varnothing$) is alternation free, hence using our favorite L^1_μ model checker we can find out that $[\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\varnothing]} = \{s_1\}$. This gives the next approximation of $X = \{s_1\}$. In the next iteration we compute

$$[\![\Phi_X]\!]_{V[X\leftarrow\{s_1\}]} = \{s_0, s_1\} \cap [\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\{s_1\}]}$$

and again use L^1_μ model checker to find $[\![\nu Y.\langle a\rangle Y \vee \langle b\rangle X]\!]_{V[X\leftarrow\{s_1\}]} = \{s_0, s_1\}$. This ends the computation since next iteration cannot add any new states to the value of X.

In general, the situation may be more complex than in our example. Several variables of maximal alternation depth may occur in a formula, possibly with different types. Variables of the same type may depend on each other. Also, if $Y \sqsubseteq X$ then we must know the value of Y before we start to compute X. Hence evaluation of these variables must be performed in bottom-up manner, using simultaneous fixed-point computations.

More precisely, let $\mathcal{X} \subseteq Var(\varphi)$ be the set of variables with maximal alternation depth. Systematically we evaluate them using the iteration method. We start with these variables in $\mathcal{X}$ which do not subsume any other variables from $\mathcal{X}$. After finding their values we proceed to the variables $X \in \mathcal{X}$ for which values of all $Y \sqsubseteq X$, $Y \in \mathcal{X}$ are already known. Eventually all variables in $\mathcal{X}$ will be evaluated.

In each stage of this process we find the values of all μ-variables and, separately, the values of all ν-variables by simultaneous iteration. Specifically, if $X_1, \ldots, X_p$ are all μ-variables which we want to evaluate, then we start by setting $V(X_i) = \varnothing$ for $i = 1, \ldots, p$, where V is the current valuation (keeping the already computed values). Then in each iteration values of $X_1, \ldots, X_p$ are updated using formula $V'(X_i) = [\![\Phi_i]\!]_V$. Here Φ_i is the formula Φ_{X_i} with all subformulae $\sigma Y.\Phi_Y$, $Y \in \mathcal{X}$ replaced by Y. Note that if $Y \sqsubseteq X_i$ for $Y \in \mathcal{X}$ then we have already computed the value of Y and it is stored in V. Observe that the alternation depth of each Φ_i is strictly less than the alternation depth of the original formula φ.

3.2 Alternation and Game Graphs

The previous idea of reducing alternation depth can be applied similarly for game graphs. To see this, we study the structure of game graphs for L_μ^2-formulae. Let us first introduce the notion of a formula's graph, which represents the subformula relation and unwindings of fixed points.

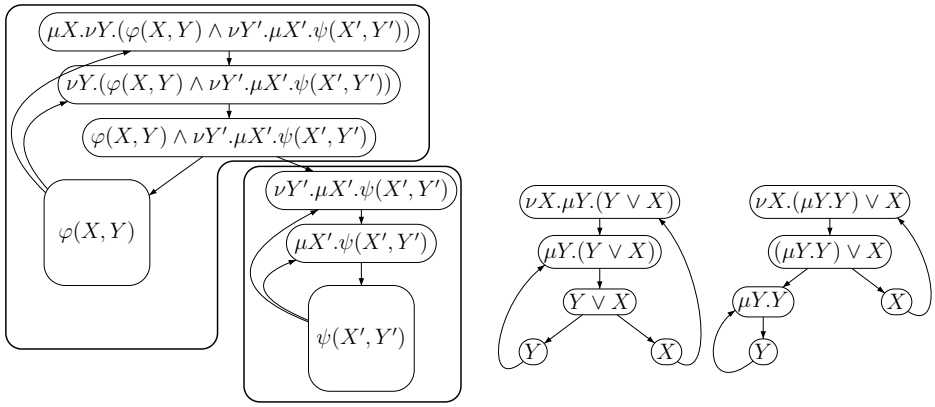
Definition 3.1 The *graph of* $\varphi \in L_\mu$, denoted by $\mathcal{G}(\varphi)$, is $(\mathrm{Sub}(\varphi), \rightarrow)$ where $\rightarrow = \{(\psi, \psi') \mid \psi'$ is a direct subformula of ψ or $\psi = X$ and $\psi' = \sigma_X.\Phi_X\}$.

Figure 2 indicates the graphs of several formulae.

An arbitrary graph can be partitioned into maximal strongly connected components and directed acyclic graphs (dags). Furthermore, these components can be partially ordered by bridges (Figure 2(a)). For the graph of an L_μ^2-formula, we can easily see, that within its strongly connected components either a μ-fixed point or ν-fixed point is unwinded (Figure 2(c)), or one alternation of μ- and ν-formulae (Figure 2(b)). We conclude:

Theorem 3.2 *Let $\varphi \in L_\mu^2$. Then there exists a partition of $\mathcal{G}(\varphi)$ such that every subgraph either is a dag, contains only variables of alternation depth one, or contains variables of alternation depth one and two that are dependent. Furthermore, the subgraphs are partially ordered by bridges.*

In Figure 2(a), we indicate the partition of the graph of the formula



(a) Partition into components (b) alternation (c) alternation-free

Fig. 2. Graphs of formulae

$\mu X.\nu Y.(\varphi(X,Y) \wedge \nu Y'.\mu X'.\psi(X',Y'))$. Here, $\varphi(X,Y)$ and $\psi(X,Y)$ are arbitrary fixed-point-free formulae with free variables among X, Y and X', Y', respectively. Note that X subsumes Y' but Y' is not dependent on X.

Since the game graph is a kind of *unfolding* of the formula graph using the transition system, the partition of the formula graph induces a partition of the game graph. We get:

Theorem 3.3 *Let $\mathcal{G}$ be the game graph for a transition system and an L^2_μ-formula. Then there exists a partition of $\mathcal{G}$ such that every subgraph is either a dag, contains only variables of alternation depth one, or variables of alternation depth two and one that are dependent. Furthermore, the subgraphs are partially ordered by bridges.*

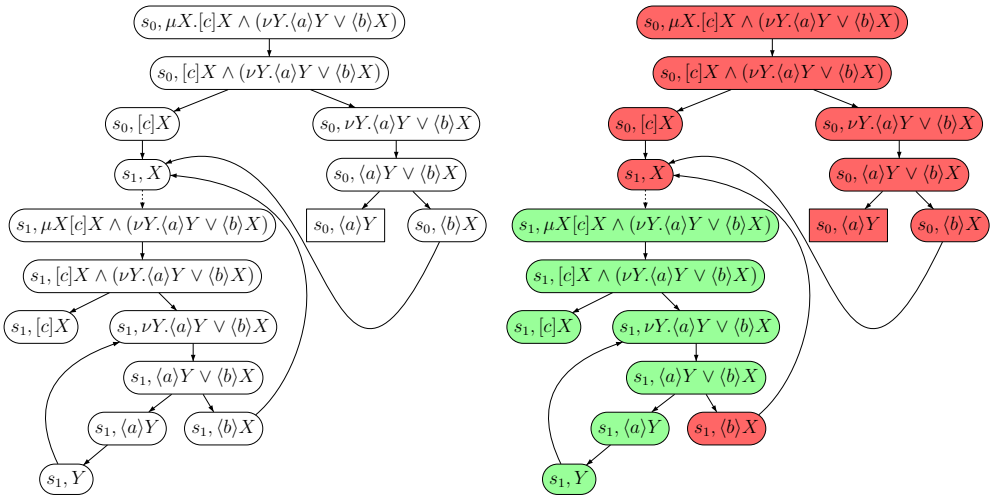
3.3 The algorithm

Let us now develop algorithm SBCL2 for finding a winning strategy, and hence solving the model checking problem for L^2_μ. The algorithm preprocesses the given game graph component-wise (using the partition according to Theorem 3.3). It employs the sequential algorithm SBCL1 presented in [3] to colour each component, possibly repeatedly with further interspersed processing steps.

The basic idea of the algorithm is to explicitly perform any fixed point calculation with alternation depth 2 directly, then reusing the algorithm SBCL1 suitable for L^1_μ to perform the remaining inner ($ad(X) = 1$) fixed point calculations. To illustrate our approach, we study the transition system shown in Figure 1 and the formula $\mu X.[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X)$.

Preprocessing the game graph.

First, we decompose the given game graph according to Theorem 3.3 into partially ordered components Q_i, which are subsequently coloured, starting



(a) preprocessing (b) one iteration

Fig. 3. Colouring a game-graph component with alternation.

with the least ones.

Let $G_i = (Q_i, E_i)$ be the graph of a component. If G_i is a dag or contains only variables of depth one, SBCL1 can be used immediately. Assume now that G_i has a variable of alternation depth 2. To simplify the presentation, we assume it is μ-variable X. The forthcoming explanation can be dualized for ν-variables.

We determine the set $PE_i \subseteq E_i$ of edges $(s, X) \to (s, \mu X.\Phi_X) \in E_i$ (for $ad(X) = 2$). The modified graph $G_i^0 = (Q_i, E_i - PE_i)$ does not admit paths containing infinitely many configurations of *both* types of fixed point formulae any more. Figure 3(a) shows the game graph for our example. The dotted edge is the one removed to obtain a game graph in which no variable of depth 2 is unwinded.

Colouring the game graph.

Next, we apply a pre-colouring of configurations q (which are now leaves in the modified graph) of edges $q \to q' \in PE_i$ according to the type of their corresponding fixed point formula—*red* for μ fixed points, *green* for ν. Then we execute algorithm SBCL1 on the pre-coloured and modified G_i^j (initially $j = 0$), which is afterwards completely coloured.

A new uncoloured graph G_i^{j+1} is created from G_i^j, and for each edge $q \to q' \in PE_i$, we copy the colour of q' in G_i^j to configuration q in G_i^{j+1}. They form the refined assumptions of the pre-colouring. SBCL1 is executed again on G_i^{j+1} afterwards.

The above step is repeated until $G_i^{j+1} = G_i^j$ and hence the fixed point is reached. The number of iterations is bounded by $j \leq |proj_1(PE_i)|$.

In our example, the configuration (s_1, X) is initialized with *red*. SBCL1

will colour the the game graph as depicted in Figure 3(b), in which *red* configurations are filled with ■ and *green* ones with ■. Since the colour of configuration $(s_1, \mu X[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X)))$ is different from (s_1, X), we set the colour of configuration (s_1, X) to *green* and start a new iteration. Now, SBCL1 will colour all configurations green (except $(s_0, \langle a\rangle Y)$). Since the colour of $(s_1, \mu X[c]X \wedge (\nu Y.\langle a\rangle Y \vee \langle b\rangle X))$ has not been changed in this iteration, we are done.

Proceeding with the remaining components is done in the same manner, the propagation of colours *between* components is the same as described in [3]. The result of SBCL2 is a completely and correctly (wrt. the rules lined out in Section 2.3) coloured game graph.

The pre-colouring preserves the property of algorithm SBCL1 that the colouring is monotonic, i.e. no configuration changes its colour more than once. Since SBCL1 terminates as well, SBCL2 is terminating. We sum up:

Theorem 3.4 *Given a transition system and an L^2_μ-formula, the algorithm SBCL2 constructs the corresponding game graph and labells the configurations either red or green, depending on whether ∀belard or respectively ∃loise has a winning strategy.*

Complexity

The run time of our colouring algorithm for L^2_μ is quadratic in both the number of states s of the underlying LTS, and in the length l of the given L^2_μ formula, thus $s^2 \times l^2$. This is straight-forward to see, recalling that the complexity of SBCL1 is bounded by $s \times l$ as stated in [3], and SBCL1 is executed at most $|\bigcup_i proj_1(PE_i)| \leq s \times l$ times.

The costs for generating the PE_i are negligible here, since it needs to be done only once (e.g. while producing the game graph), and is linear wrt. to the number of edges in $\mathcal{G}$.

4 Parallel Model Checking

We describe a parallelized version of SBCL2, called PTCL2. We line out how the game graph data structure can be built and distributed in parallel. The key idea of the parallelized colouring process is again to reduce the alternation depth by calculating the outermost fixed point explicitly, then executing a parallel algorithm for L^1_μ on the transformed game graph.

4.1 Distributing the game graph

The parallel construction and distribution of the game graph is a standard approach, and basically the same as in [3], which we are going to revisit shortly.

As a data structure, we employ adjacency lists. We also link to the predecessor, as well as to the successor of a configuration for the labelling al-

gorithm. A component is constructed in parallel by a typical breadth-first strategy: given a configuration q, determine its successors $q_1, \ldots, q_n$. To obtain a deterministic distribution of the configurations over the workstation cluster, one takes a function in the spirit of a hash function assigning to every configuration an integer and subsequently its value modulo N, the number of processors. This function $f : (S \times \mathrm{Sub}(\varphi)) \to \{0, \ldots, N-1\}$ determines the location of every configuration within the network uniquely and without global knowledge. Thus, we can send each $q \in \{q_1, \ldots, q_n\}$ 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 successors. These are sent in the same manner to their owning (wrt. f) processors, together with the information that q is a predecessor. The corresponding processors update their local memory similarly.

Additionally, we also take care of the sets PE_i during the construction of the game graph. Just as in the sequential case, all edges in PE_i are removed from the game graph, and instead are stored separately on processor 0, which we hereafter refer to as "master". In practice, their number is sufficiently small that this approach neither leads to space problems nor processing bottlenecks.

Furthermore, the distribution function f is modified in such a way that each configuration $q \in \mathit{proj}_2(\bigcup_i PE_i)$ (hereafter called "iteration configuration"), is stored on the master, thus $f(q) = 0$. The reason is that the master takes special actions on a colour change of such an iteration configuration, which we will outline below.

We refer the reader to [2] for a thorough discussion of this and other possible approaches storing distributed transition systems.

4.2 Labelling in parallel

Due to our game graph's construction, it is devoid of any *SCC*s created due to a L^2_μ formula, and hence we can apply our standard parallel model checking algorithm for L^1_μ formulae (PTCL1, from [3, Section 4]) to colour it. Again, the colouring is done component-wise and the propagation of colours between components can be taken verbatim from PTCL1.

Similar to the sequential case, PTCL1 is called repeatedly on a component Q_i, as long as a run yields a colour change on any of the iteration configurations. Since the master owns all iteration configurations, it can easily check after each round of colouring for such a colour change of a configuration q'. The colouring process is then restarted as follows.

The master notifies all processors (including itself) to uncolour their parts of the game graph, the corresponding predecessor q (wrt. PE_i) of any q' adjusts its initial colouring to the colour of q'. Then all processors are notified by a broadcast from the master to restart PTCL1.

If no iteration configuration has changed its colour after applying PTCL1, the colouring is stable (the fixed point has been reached), and the algorithm terminates, returning as result the colour of the root configuration of $\mathscr{G}$, and thus the solution to the model checking problem.

5 Conclusion

In this paper, we have presented a *parallel* game-based model-checking algorithm for an important fragment of the μ-calculus. It immediately gives parallel model checking algorithms for LTL and CTL* as well. The idea of the algorithm is to reduce alternation depth and to employ the parallel model checking algorithm presented in [3].

It would be desirable to have an implementation of our algorithm to certify its practical benefits. Furthermore, it would be interesting to study its performance in comparison with existing parallel model checking algorithm for the μ-calculus and LTL.

References

[1] J. Barnat, L. Brim, and I. Černá. Property driven distribution of nested DFS. In *VCL 2002: The Third International Workshop on Verification and Computational Logic, Pittsburgh PA, October 5, 2002 (held at the PLI 2002 Symposium)*, 2002.

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

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

[4] J. C. Bradfield. The modal mu-calculus alternation hierarchy is strict. In U. Montanari and V. Sassone, editors, *CONCUR'96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 233–246, Pisa, Italy, 26–29 Aug. 1996. Springer.

[5] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model-checking based on negative cycle detection. In *Proceedings of 21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'01)*, Lecture Notes in Computer Science. Springer, Dec. 2001.

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

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

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

[9] 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, July 2001.

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

[11] 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, June 1997. Haifa, Israel, June 22-25.

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