Chapter 12
Parallel Model Checking Algorithms for Linear-time Temporal Logic

Jiri Barnat, Vincent Bloemen, Alexandre Duret-Lutz, Alfons Laarman, Laure Petrucci, Jaco van de Pol, and Etienne Renault

Abstract

Model checking is a fully automated, formal method for demonstrating absence of bugs in reactive systems. Here, bugs are violations of properties in Linear-time Temporal Logic (LTL). A fundamental challenge to its application is the exponential explosion in the number of system states. The current chapter discusses the use of parallelism in order to overcome this challenge.

We reiterate the textbook automata-theoretic approach, which reduces the model checking problem to the graph problem of finding cycles. We discuss several parallel algorithms that attack this problem in various ways, each with different characteristics: Depth-first search (DFS) based algorithms rely on heuristics for good parallelization, but exhibit a low complexity and good on-the-fly behavior. Breadth-first search (BFS) based approaches, on the other hand, offer good parallel scalability and support distributed parallelism.

In addition, we present various simpler model checking tasks, which still solve a large and important subset of the LTL model checking problem, and

Jiri Barnat
Masaryk University, Brno, Czech Republic, e-mail: xbarnat@fi.muni.cz

Vincent Bloemen
University of Twente, Enschede, The Netherlands, e-mail: v.bloemen@utwente.nl

Alexandre Duret-Lutz
LRDE, Epita, Paris, France, e-mail: adl@lrde.epita.fr

Alfons Laarman
Leiden University, Leiden, The Netherlands, e-mail: a.w.laarman@liacs.leidenuniv.nl

Laure Petrucci
LIPN, CNRS, Paris, France, e-mail: Laure.Petrucci@lipn.univ-paris13.fr

Jaco van de Pol
University of Twente, Enschede, The Netherlands, e-mail: j.c.vandepol@utwente.nl

Etienne Renault
LRDE, Epita, Paris, France, e-mail: renault@lrde.epita.fr

show how these can be exploited to yield more efficient algorithms. In particular, we provide simplified DFS-based search algorithms and show that the BFS-based algorithms exhibit optimal runtimes in certain cases.

12.1 Introduction

This chapter discusses parallel algorithms for model checking properties of Linear-time Temporal Logic (LTL). Model checking [30, 8] is a verification technique to establish the correctness of hardware and software systems. In contrast to theorem proving, model checking is a fully automated procedure, invented by the Turing Award winners Clarke, Emerson, and Sifakis (2007). In contrast to testing, it is a complete and exhaustive method. Nowadays, along with testing and static analysis, model checking is an indispensable industrial tool for eliminating bugs and increasing confidence in hardware designs (e.g., at Intel [45] and IBM [15]) and software products (e.g., at Microsoft [9]). For an example case study, refer to Chapter 16, An Application of Parallel Satisfiability Solving to the Verification of Complex Embedded Systems.

Formally, model checking solves the problem: "Does model M satisfy property P?" ($M \vDash P$). Here the model M is a finite abstraction of a hardware or software system, provided in the form of a transition system. The paths in the graph of model M consist of infinite sequences of states connected by state transitions. Paths correspond to possible runs of the system. The property P is specified in some temporal logic. In this chapter, we restrict the discussion to Linear-time Temporal Logic (LTL). An LTL property denotes a set of paths, so P can be viewed as a specification of the correct runs of the system. Section 12.2 will formalize the syntax and semantics of LTL and identify some important fragments. For this introduction, it is sufficient to view model checking as a graph search problem, where the goal is to find a bad state or, more generally, a cycle representing an infinite path violating the property.

The main obstacle to model checking is the size of the transition system, often referred to as "the state space explosion" [96]. This graph grows exponentially in the number of components and variables in the specification, mainly due to parallel interleaving in concurrent systems, and the Cartesian product of data domains. Many sequential algorithms exist to address the state space explosion, reducing the state space by exploiting symmetries [29, 41, 20, 61], restricting the interleavings to be checked [95, 63, 54, 1], or abstracting the data domains [27, 24]. Another direction is to represent state spaces symbolically, applying powerful techniques such as Binary Decision Diagrams (BDD) [23, 77] or satisfiability (SAT) [28, 16, 78]. Parallel satisfiability is discussed in Chapter 1, Parallel Satisfiability, and parallel decision diagrams in Chapter 13, Multi-core Decision Diagrams. Although these methods greatly reduce the memory and time usage of model checking, the ever-growing complexity of hardware and software designs has meant that, so far, the practical application of model checking is still hindered by memory and time resources.

Parallel model checking algorithms — pragmatics

This chapter focuses on recent advances in utilizing more hardware resources to solve the model checking problem. In distributed model checking, the memory problem is alleviated by distributing the state space over the memory of many computers in some network (cluster, cloud). Recently, several new approaches to parallel model checking emerged using multiple processors in a shared-memory machine to speed up model checking computations. Both approaches are highly non-trivial, since graph (search) algorithms must be redesigned to be fit for parallel computation. Next, we consider parallel graph algorithms from pragmatic and theoretical points of view.

From a pragmatic point of view, obtaining good parallel speedups for graph problems is notoriously hard [75, 73]. This is mainly caused by the irregularity of graphs. The efficiency of parallel programs often depends on exploiting locality, which can be predicted for regular data structures like matrices. However, state spaces are irregular sparse graphs, whose shape highly depends on the model at hand. For distributed algorithms, the consequence is that traversing a transition from a source state in the graph often requires communication with the machine where the target is stored, leading to a dramatic communication overhead. For multi-core computing, the threads are continually looking up the location of target states in main memory. Since main memory (and the memory bus) are a shared resource, memory-intensive algorithms are hard to speed up on multi-core machines. As a consequence, practical implementations pay a lot of attention to low-level details, such as local caching, evading the need for locks using atomic instructions such as compare-and-swap, and latency hiding by asynchronous communication. This chapter does not focus on these implementation details, although they are essential to demonstrate that the treated algorithms achieve speedup in practice.

Instead, we focus on the algorithmic aspects. We review the basic sequential algorithms for LTL model checking in Section 12.3. These subproblems can be solved by linear-time algorithms. However, today the only known linear time algorithms heavily depend on the Depth-First Search (DFS) strategy, which (as we will explain below) is hard to parallelize. This holds for LTL algorithms based on Nested Depth-First Search as well as for those based on the analysis of the Strongly Connected Components.

Another reason for our general preference for DFS lies in the nature of search. If we use the algorithm to search for bugs (bug hunting), we can terminate as soon as the first bug has been found. It would be a waste of resources if we were to first compute the whole state space and then search only a small part to find the bug. The DFS-based algorithms are generally well-suited for on-the-fly model checking, where computing the state space and checking the properties are intertwined. This carries over to parallel search. It is well known that parallel random search can achieve superlinear speedups when the goal states are uniformly distributed [81, 72]. In case of full verification of programs, this consideration is less important. We will present parallel DFS-based algorithms in Section 12.4.

Parallel model checking algorithms — theoretical considerations

Finally, what does theory actually say? A well-known result [82] is that DFS is inherently sequential. To understand this, we recall the class NC (Nick's Class) of problems that admit scalable parallel algorithms [56]: A problem is in NC if it can be solved in poly-logarithmic time $(\log n)^{o(1)}$ using a polynomial amount of hardware, i.e., $n^{o(1)}$ processors. Let P be the class of problems that admit a polynomial-time algorithm. A problem is P-complete, if all problems in P can be reduced to it by an NC algorithm. The canonical P-complete problem is CVP, the circuit valuation problem (given a circuit, and its Boolean input values, determine the value of its output). Although formally open, it is widely believed that NC does not contain P-complete problems, so problems in P are "inherently sequential." Note that if NC contained a single P-complete problem, then all polynomial problems would be parallelizable. Reif [82] actually showed that lexicographic DFS is P-complete by a direct reduction to the CVP. Hence, given a graph and a fixed ordering of transitions from each state, there is probably no parallel algorithm to even check whether node x will be visited before node y in the DFS post-order, observing the fixed transition ordering.

The following intellectual positions are possible in relation to this fact from theory: First, one can decide to ignore this theoretical restriction. This is the position in Section 12.4. We introduce various parallel random DFS algorithms for which we have shown practical speedup, even though they are not poly-logarithmic. The main motivation is that, in practice, the number of processors is much smaller than the size of the graph. A practical speedup for graphs from 10^3 to 10^8 nodes does not contradict the impossibility result in the limit case of $(10^8)^k$ processors.

The second position is to take the theoretical result seriously, and avoid DFS algorithms. Parallel BFS (breadth-first search), and hence SCC decomposition, is in NC [51], which can be shown by computing transitive closure with matrix multiplication. Several BFS-based model checking algorithms and SCC decomposition methods have been designed. Although their worst-case time complexity is strictly more than linear, they behave well on practical instances, and are even linear for many model checking fragments. Moreover, since BFS-based algorithms can be parallelized, with sufficiently many processors this approach should scale (even though the increased work-complexity doesn't admit a provably efficient parallel solution). Algorithms OWCTY and MAP in Section 12.5 are an illustration of BFS-based algorithms in this category.

The third possibility is to circumvent the theoretical results. Note that it is technically still possible that non-lexicographic DFS (without fixing the ordering of the transitions in advance) is in NC. Actually, it has been proved that free DFS is in NC indeed for planar graphs [57], and for general graphs the problem is known to be in Random NC [3]. We do not claim complexity-theoretic results in this chapter. Our random parallel free DFS algorithms will not provide a single global post-order and, in the worst case, they don't run in parallel logarithmic time. However, we have proved that they provide sufficient ordering to solve the model checking problem, and we demonstrated have good speedups for practical problems. Eventually, this approach might shed some light on this intriguing 30-year-old open problem.

12.2 Preliminaries: LTL Model Checking and Automata

The current section explains the theoretical foundation of LTL model checking. The formal approach taken here is to interpret both the system and its specification as an automaton. We will show that this automaton is exponential in the size of both the system and the specification and develop the constructs required by the LTL model checking algorithms in the subsequent section to efficiently handle such large automata.

12.2.1 Automata-Theoretic Model Checking

Model checkers are tools that take two inputs: some model M of a system, and some specification φ that should be satisfied by all possible behaviors of M. For instance if M is a model of a road intersection with traffic lights and sensors, the property φ could specify that whenever a car is sensed the light of its lane should eventually become green. Note that such a property is not necessarily about the *state* of the system: in this example it is about its possible behaviors, i.e., the evolution of its state. Furthermore, the behaviors of this system are infinite.

Model checking [97] decides whether some model M satisfies some specification φ (which we denote $M \models \varphi$). In the automata-theoretic approach, the model M is first converted into an automaton K_M whose language $\mathscr{L}(K_M)$ represents the set of all (infinite) behaviors of M. The negation of the formula φ is converted into an automaton $A_{\neg\varphi}$ whose language $\mathscr{L}(A_{\neg\varphi})$ captures the forbidden behaviors. With these objects, testing whether M satisfies φ

amounts to checking the emptiness of the product of the two automata: if $\mathscr{L}(K_M \otimes A_{\neg\varphi}) = \emptyset$, then $M \models \varphi$. If $\mathscr{L}(K_M \otimes A_{\neg\varphi})$ is found not to be empty, it means there exists a counterexample: a behavior of M that invalidates φ.

12.2.2 Sequences and ω-Words

We shall use $\mathbb{B} = \{\bot, \top\}$ to denote the set of Boolean values, $\omega = \{0, 1, 2, \ldots\}$ for the set of non-negative integers, and $[n] = \{0, 1, 2, \ldots, n-1\}$ the first n of those. By convention $[0] = \emptyset$.

Let AP be a finite set of (atomic) propositions. An *assignment* is a function $x : \mathrm{AP} \to \mathbb{B}$ that evaluates each proposition. We use $\mathbb{B}^{\mathrm{AP}}$ to denote the set of all assignments of AP.

An *infinite sequence* over some set Σ is a function $\sigma : \omega \to \Sigma$ that assigns an element of Σ to each possible index. We use Σ^ω to denote the set of infinite sequences over Σ.

A *finite sequence* of length n over Σ is a function $\sigma : [n] \to \Sigma$. We use Σ^* for the set of all finite sequences of any length $n \geq 0$, and Σ^+ for the set of finite sequences of length $n > 0$.

To define a particular sequence, we denote it by the concatenation of its elements $x_i \in \Sigma$ as $\sigma = x_0; x_1; x_2; \ldots$, meaning that $\sigma(i) = x_i$.

For some infinite sequence $\sigma \in \Sigma^\omega$, we use σ^i to denote the sequence obtained from σ by removing its first $i \geq 0$ elements; i.e., $\sigma^i(j) = \sigma(i+j)$ for all j. We denote by $\mathrm{Inf}(\sigma) \subseteq \Sigma$ the set of elements that appear infinitely often in σ, i.e., $\mathrm{Inf}(\sigma) = \{s \in \Sigma \mid \forall i \in \omega, \exists j > i, \sigma(j) = s\}$.

In this chapter AP is assumed to be fixed, and infinite sequences of assignments, i.e., elements of $(\mathbb{B}^{\mathrm{AP}})^\omega$, are called *ω-words*. Finally, a *language* is a (possibly infinite) set of ω-words.

12.2.3 Linear-time Temporal Logic

In model checking, ω-words are used to represent the different behaviors of the system to check.

Linear-time Temporal Logic (LTL) formulas are typically used to specify the property to verify on the system by specifying which ω-words should be accepted or rejected. LTL formulas are constructed according to the following grammar, where $a \in \mathrm{AP}$:

$$\varphi ::= \top \mid \bot \mid a \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \mathbin{\mathsf{U}} \varphi \mid \varphi \mathbin{\mathsf{R}} \varphi \mid \mathsf{F}\varphi \mid \mathsf{G}\varphi \mid \mathsf{X}\varphi$$

Given an ω-word $\sigma \in (\mathbb{B}^{\mathrm{AP}})^\omega$ and an LTL formula φ, we say that σ satisfies φ (denoted $\sigma \models \varphi$) according to the following semantics. For any $a \in \mathrm{AP}$ and any LTL formulas φ_1 and φ_2,

$$\begin{array}{ll}
\sigma \models \top & \\
\sigma \not\models \bot & \\
\sigma \models a & \text{iff } \sigma(0)(a) = \top \\
\sigma \models \neg\varphi_1 & \text{iff } \sigma \not\models \varphi_1 \\
\sigma \models \varphi_1 \lor \varphi_2 & \text{iff } (\sigma \models \varphi_1) \lor (\sigma \models \varphi_2) \\
\sigma \models \varphi_1 \land \varphi_2 & \text{iff } (\sigma \models \varphi_1) \land (\sigma \models \varphi_2) \\
\sigma \models \varphi_1 \mathbin{\mathsf{U}} \varphi_2 & \text{iff } \exists i \geq 0, (\sigma^i \models \varphi_2) \land (\forall j < i,\, \sigma^j \models \varphi_1) \\
\sigma \models \varphi_1 \mathbin{\mathsf{R}} \varphi_2 & \text{iff } \forall i \geq 0, (\sigma^i \models \varphi_2) \lor (\exists j < i,\, \sigma^j \models \varphi_1) \\
\sigma \models \mathsf{F}\varphi_1 & \text{iff } \exists i \geq 0,\, \sigma^i \models \varphi_1 \\
\sigma \models \mathsf{G}\varphi_1 & \text{iff } \forall i \geq 0,\, \sigma^i \models \varphi_1 \\
\sigma \models \mathsf{X}\varphi_1 & \text{iff } \sigma^1 \models \varphi_1
\end{array}$$

The language of a formula φ is the set of words that satisfy it: $\mathscr{L}(\varphi) = \{\sigma \in (\mathbb{B}^{\mathrm{AP}})^\omega \mid \sigma \models \varphi\}$. Two LTL formulas are *equivalent* iff they have the same language: $\varphi_1 \equiv \varphi_2 \iff \mathscr{L}(\varphi_1) = \mathscr{L}(\varphi_2)$. For example one can see that $\neg\mathsf{FG}a \equiv \mathsf{GF}\neg a$.

The size of an LTL formula φ, denoted $|\varphi|$, is the number of symbols in φ. For example $|\neg\mathsf{FG}a| = 4$.

12.2.4 Kripke Structures

A Kripke structure is an automaton with states labeled by assignments.

Definition 1 (Kripke Structure). A Kripke structure is a tuple $K = (Q, \iota, \delta, \ell)$ where

- Q is a finite set of states,
- $\iota \in Q$ is the initial state,
- $\delta \subseteq Q \times Q$ is a set of transitions,
- $\ell : Q \to \mathbb{B}^{\mathrm{AP}}$ is a function labeling each state with an assignment.

The runs of K, denoted $\mathrm{Runs}(K)$, are the infinite sequences of states $\rho \in Q^\omega$ that start with ι and follow transitions in δ:

$$\mathrm{Runs}(K) = \{\rho \in Q^\omega \mid \rho(0) = \iota \text{ and } \forall i \geq 0,\, (\rho(i), \rho(i+1)) \in \delta\}$$

If we naturally extend the labeling function ℓ to runs, then each run ρ is associated with an ω-word $\ell(\rho)$ defined by $\ell(\rho)(i) = \ell(\rho(i))$. The language $\mathscr{L}(K)$ of the Kripke structure is the set of words associated with all its runs: $\mathscr{L}(K) = \{\ell(\rho) \mid \rho \in \mathrm{Runs}(K)\}$.

Definition 2 (Deadlock-Free Kripke Structure). A Kripke structure is said to be *deadlock-free* if all its states have at least one successor. In other words $K = (Q, \iota, \delta, \ell)$ is *deadlock-free* if $\forall s \in Q,\, \exists d \in Q,\, (s, d) \in \delta$.

12.2.5 Büchi Automata

Büchi automata can represent ω-regular languages. We shall define different flavors of Büchi automata that correspond to combinations of the following two options:

- transition-based or state-based acceptance
- classical Büchi acceptance, or generalized Büchi acceptance.

While all the resulting automata have the same expressive power, they can have different degrees of conciseness, and may require different emptiness-check procedures. Hence from the model checking point of view, these choices can affect memory consumption and emptiness-check complexity.

Definition 3 (TGBA). A *Transition-based Generalized Büchi Automaton* is a tuple $A = (Q, \iota, \delta, n, M)$ where

- Q is a finite set of states,
- $\iota \in Q$ is the initial state,
- $\delta \subseteq Q \times \mathbb{B}^{\mathrm{AP}} \times Q$ is a set of transitions,
- n is an integer specifying a number of accepting marks,
- $M : \delta \to 2^{[n]}$ is a *marking* function that specifies a subset of marks associated with each transition.

For a transition $t \in \delta$ we write t^s for its source, t^ℓ for its label, and t^d for its destination: $t = (t^s, t^\ell, t^d)$.

The runs of A are infinite sequences of consecutive transitions:

$$\mathrm{Runs}(A) = \{\rho \in \delta^\omega \mid \rho(0)^s = \iota \text{ and } \forall i \geq 0,\ \rho(i)^d = \rho(i+1)^s\}$$

The *accepting runs* of A are those that have, for each acceptance mark, infinitely many transitions with that mark:

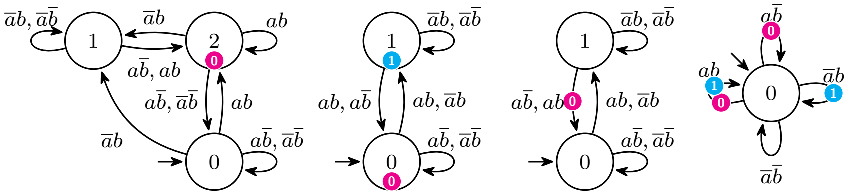
$$\mathrm{Acc}(A) = \left\{\rho \in \mathrm{Runs}(A) \;\middle|\; [n] = \bigcup_{t \in \mathrm{Inf}(\rho)} M(t)\right\}$$

Let us also define the word $\ell(\rho)$ associated with a run ρ by $\ell(\rho)(i) = \rho(i)^\ell$. Now the language $\mathscr{L}(A)$ of the automaton A is the set of words associated with its accepting runs:

$$\mathscr{L}(A) = \{\ell(\rho) \mid \rho \in \mathrm{Acc}(A)\}$$

For convenience, we will also overload the δ notation and write $\delta(q)$ for the set of outgoing transitions of any state $q \in Q$: $\delta(q) = \{(s, x, d) \in \delta \mid s = q\}$.

Definition 4 (SGBA). A *State-based Generalized Büchi Automaton* is also a tuple $A = (Q, \iota, \delta, n, M)$, with identical definitions for Q, ι, δ, and n, but this time the marking function M associates marks with states: $M : Q \to 2^{[n]}$. The


$\bar{a}b, \bar{a}\bar{b}$
1
$\bar{a}b$
2
0
ab
$a\bar{b}, ab$
$a\bar{b}, \bar{a}\bar{b}$
ab
$\bar{a}b$
0
$a\bar{b}, \bar{a}\bar{b}$
1
1
$\bar{a}b, \bar{a}\bar{b}$
$ab, a\bar{b}$
$ab, \bar{a}b$
0
0
$a\bar{b}, \bar{a}\bar{b}$
1
$\bar{a}b, \bar{a}\bar{b}$
$a\bar{b}, ab$
0
$ab, \bar{a}b$
0
$a\bar{b}, \bar{a}\bar{b}$
$a\bar{b}$
0
ab
1
0
0
$\bar{a}b$
1
$\bar{a}\bar{b}$

(a) SBA (b) SGBA (c) TBA (d) TGBA

Fig. 12.1 Minimal deterministic automata recognizing $\mathscr{L}(\mathsf{GF}a \land \mathsf{GF}b)$. The SGBA and TGBA use $n = 2$ accepting marks, while the SBA and TBA have $n = 1$ by definition

runs are defined similarly. The accepting runs are those that have infinitely many states marked with each acceptance mark:

$$\mathrm{Acc}(A) = \left\{ \rho \in \mathrm{Runs}(A) \;\middle|\; [n] = \bigcup_{t \in \mathrm{Inf}(\rho)} M(t^s) \right\}$$

and then the automaton's language is still defined as $\mathscr{L}(A) = \{\ell(\rho) \mid \rho \in \mathrm{Acc}(A)\}$.

Definition 5 (SBA and TBA). *State-based* and *Transition-based Büchi Automata* are particular cases of the above definitions where $n = 1$.

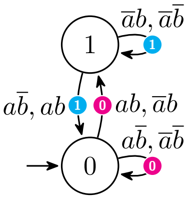


Fig. 12.2 How to interpret the SGBA of Fig. 12.1(b) as a TGBA

Figure 12.1 shows four automata with different acceptance conditions, all recognizing the language of the LTL formula $\mathsf{GF}a \land \mathsf{GF}b$: a and b should each hold infinitely often, but not necessary at the same time. As usual, multiple transitions of the form (s, x, d) and (s, y, d) are pictured as a single edge $\textcircled{s} \xrightarrow{x,y} \textcircled{d}$. Marked states and transitions are denoted using colored bullets such as ⓿ or ❶. So the fact that $M\big((s, x, d)\big) = \{0, 1\}$ is pictured as $\textcircled{s}$-⓿$\xrightarrow{x}$❶$\rightarrow\textcircled{d}$. Looking at the automaton of Figure 12.1(b), the run $\rho_1 = (0, \bar{a}b, 1); (1, \bar{a}b, 1); (1, a\bar{b}, 0); (0, \bar{a}b, 1); (1, \bar{a}b, 1); (1, a\bar{b}, 0); \ldots$ is an accepting run for the word $\bar{a}b; \bar{a}b; a\bar{b}; \bar{a}b; \bar{a}b; a\bar{b}; \ldots$ as it visits ⓿ and ❶ infinitely often. The run $\rho_2 = (0, \bar{a}b, 1); (1, \bar{a}b, 1); (1, \bar{a}b, 1); (1, \bar{a}b, 1); \ldots$ is not accepting because it only visits ❶ infinitely often. By comparing the two definitions of Acc, it is clear that an SGBA $A = (Q, \iota, \delta, n, M)$ can be converted into a language-equivalent TGBA $B = (Q, \iota, \delta, n, M')$ by defining $M'(t) = M(t^s)$. This amounts to pushing the acceptance marks onto the outgoing transitions, as in Figure 12.2.

The automata of Figure 12.1 are minimal in the sense that there does not exist language-equivalent automata with the same acceptance condition and fewer states. This figure is therefore an example showing how TGBAs *can be* more concise than the other types of automata presented, but in Section 12.2.8 we will also discuss some classes of properties for which using SBAs is sufficient, i.e., no reduction can be obtained by using generalized or transition-based acceptance.

Property 1. Any TGBA (Q, ι, δ, n, M) can be "degeneralized" into a language-equivalent SBA with at most $(n+1)\,|Q|$ states, or into a language-equivalent TBA with at most $n \cdot |Q|$ states.

There exist several variants of degeneralization constructions, discussed for instance by Gastin and Oddoux [49], or Giannakopoulou and Lerda [53], and improved by Babiak et al. [7]. The automata of Figures 12.1(a) and (c) are typically what one could obtain by degeneralizing the TGBA of Figure 12.1(d).

Property 2. For any LTL formula φ, there exists a language-equivalent TGBA with $\mathrm{O}(2^{|\varphi|})$ states and $n = \mathrm{O}(|\varphi|)$ acceptance marks.

Numerous *translations* from LTL to TGBAs exist, and are implemented in tools such as `ltl2ba` [49], `ltl3ba` [6], or Spot's `ltl2tgba` [37]. Now, combining Properties 1 and 2, we get

Property 3. For any LTL formula φ, there exists a language-equivalent SBA with $\mathrm{O}(|\varphi| \cdot 2^{|\varphi|})$ states.

These upper bounds are rarely reached in practice. For instance Dwyer et al. [39] define 55 LTL formulas[1] that represent 11 intents (Absence, Response, Precedence, etc) combined with five different scopes (Before, Between, After, etc). These 55 formulas have an average size of 16.75 (maximum 40), but the SBAs produced by `ltl2tgba` (from Spot 2.1) have on average only 3.945 states (maximum 13). Using TGBAs instead of SBAs is only marginally better: `ltl2tgba` produces TGBAs with an average of 3.782 states (maximum 10); we will discuss this point in Section 12.2.8.

These small automata, representing the negation of a property we want to check, will be combined with a (potentially very large) Kripke structure representing the state space of the model to verify.

Property 4 (Synchronized product). Let $K = (Q_1, \iota_1, \delta_1, \ell)$ be a Kripke structure, and $A = (Q_2, \iota_2, \delta_2, n, M)$ be a TGBA. Then the TGBA $K \otimes A = (Q', \iota', \delta', n, M')$ where

- $Q' = Q_1 \times Q_2$,
- $\iota' = (\iota_1, \iota_2)$,
- $((s_1, s_2), x, (d_1, d_2)) \in \delta' \iff (s_1, d_1) \in \delta_1 \wedge \ell(s_1) = x \wedge (s_2, x, d_2) \in \delta_2$,
- $M'\big(((s_1, s_2), x, (d_1, d_2))\big) = M\big((s_2, x, d_2)\big)$,

is such that $\mathscr{L}(K \otimes A) = \mathscr{L}(K) \cap \mathscr{L}(A)$.

[1] http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml

The product between a Kripke structure and a SGBA can be defined similarly, with $M'\big((s_1, s_2)\big) = M(s_2)$ as the only change.

Clearly $|Q'| = |Q_1| \cdot |Q_2|$. However the states reachable from ι' can be a subset of that, and only that subset needs to be explored to decide whether $\mathscr{L}(K \otimes A)$ is empty.

12.2.6 The Emptiness-Check Problem

The emptiness-check problem can be presented as follows:

Given an automaton $B = (Q, \iota, \delta, n, M)$, decide whether $\mathscr{L}(B) = \emptyset$.

The automaton B could be any type of automaton presented previously. We will focus on TGBA, the more compact ones, as well as SBA, more frequently used because of their simple structure.

Property 5. If $\mathscr{L}(B) \neq \emptyset$, then there exists a *lasso-shaped* accepting run, i.e., a run $\rho \in \mathrm{Acc}(B)$ for which there exist $i \geq 0$ and $j \geq i$ such that $\rho(i) = \rho(j)$. (Figure 12.3.)

To show the existence of such a run, consider an automaton B (a TGBA or SGBA) and assume that $\mathscr{L}(B) \neq \emptyset$. Then by definition of $\mathscr{L}(B)$, there exists an accepting run $\pi \in \mathrm{Acc}(B)$, but that run is not necessarily lasso-shaped. The set $\mathrm{Inf}(\pi)$ contains transitions of B that (1) are visited infinitely often by π, (2) cover all acceptance marks (since π is accepting), (3) are all reachable from one another, and (4) are reachable from the initial state. Then a lasso-shaped run ρ can be constructed by building a prefix connecting the initial state of B to any transition $t \in \mathrm{Inf}(\pi)$, and then building a cycle around t that visits all transitions of $\mathrm{Inf}(\pi)$. Note that for the lasso-shaped run ρ, the set $\mathrm{Inf}(\rho)$ corresponds exactly to the transitions that appear on the cycle. We therefore have $\mathrm{Inf}(\rho) \supseteq \mathrm{Inf}(\pi)$, which entails that ρ is also accepting.

Definition 6 (Accepting cycle). Given a TGBA (Q, ι, δ, n, M), and a finite sequence of transitions $c \in \delta^+$ of length k. We say that c is a *cycle* if its transitions actually form a cycle: $\forall i < k$, $c(i)^d = c(i + 1 \mod k)^s$.

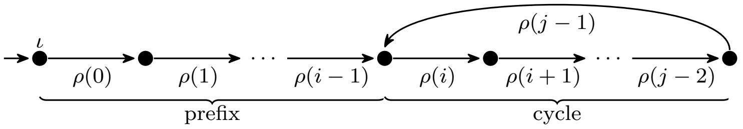


Fig. 12.3 A lasso-shaped run can be built from two finite sequences of transitions: a (possibly empty) prefix and a (non-empty) cycle

We say that a cycle c is an *elementary cycle* if additionally $|\{c(i)^s \mid i < k\}| = k$, i.e., if c goes through k different states.

We say that a cycle c is an *accepting cycle* if its transitions visit each acceptance mark at least once: $\forall i \in [n], \exists j < k, i \in M(c(j))$. Accepting cycles for SGBA are defined likewise, replacing $M(c(i))$ by $M(c(i)^s)$.

Note that the cycle part of any lasso-shaped accepting run is an accepting cycle. Combining this with Property 5 allows us to reduce the emptiness-check problem to the search for an accepting cycle.

Property 6. For an automaton B, we have $\mathscr{L}(B) \neq \emptyset$ if and only if B contains an accepting cycle reachable from the initial state.

However the number of cycles can be infinite, so it is useful to consider the simpler case where only *elementary cycles* need to be checked for acceptance:

Property 7. For an automaton B with $n \leq 1$ acceptance marks, we have $\mathscr{L}(B) \neq \emptyset$ if and only if B contains an accepting *elementary* cycle reachable from the initial state.

The case with $n = 0$ is obvious, since any cycle would be accepting, and if a cycle exists, an elementary cycle also exists. For $n = 1$, any accepting cycle c contains some transition $c(i)$ such that $M(c(i)) = 1$, and there necessarily exists some elementary accepting cycle around this transition. Note that this does not hold for $n \geq 2$, as in the example of Figure 12.4 where the only two elementary cycles are rejecting, but they can be combined to form an infinite number of accepting cycles.

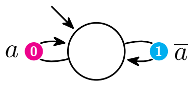


Fig. 12.4 This TGBA has a infinite number of accepting cycles; none are elementary

The goal of all emptiness-check algorithms presented in the sequel is to establish the existence or absence of such accepting cycles. Finding an accepting lasso-shaped run is one direct way to prove the existence of a reachable accepting cycle, but it is not the only one. Another one, which is especially useful with generalized acceptance ($n \geq 2$), is to prove that the automaton has a (reachable) strongly connected component that covers all acceptance marks. This is formalized by Definition 7 and Property 8.

Definition 7 (SCC). In an automaton (Q, ι, δ, n, M), a *partial strongly connected component* (partial SCC) is a nonempty set of states $C \subseteq Q$ such that any ordered pair of states of C can be connected by a sequence of consecutive transitions. If additionally C is maximal with respect to set inclusion, we call it a *maximal strongly connected component* (maximal SCC). Let us use $C_\delta = \{(s, x, d) \in \delta \mid s \in C, d \in C\}$ to denote the set of transitions induced by C.

We call an SCC C *trivial* if $C_\delta = \emptyset$. In a TGBA we say that a non-trivial SCC C is accepting if C_δ covers all acceptance marks, i.e., $\forall i \in [n]$, $\exists t \in$

C_δ, $i \in M(t)$. In an SGBA a non-trivial SCC C is accepting if C covers all acceptance marks, i.e., $\forall i \in [n]$, $\exists s \in C$, $i \in M(s)$.

A *rejecting* SCC is either a trivial SCC, or a non-trivial SCC that does not cover all acceptance marks.

Property 8. For an automaton B, we have $\mathscr{L}(B) \neq \emptyset$ if and only if the initial state can reach an accepting SCC.

Note that it does not matter whether the accepting SCC is partial. SCC-based emptiness checks usually maintain a set of partial SCCs, to which they add new states when cycles are discovered. For each (reachable) partial SCC C they maintain the set of acceptance marks seen in C (that is $S_C = \bigcup_{t \in C_\delta} M(t)$ in the case of TGBAs, or $S_C = \bigcup_{s \in C} M(s)$ for SGBAs), and they can report the non-emptiness of the automaton as soon as one of these sets equals $[n]$.

In the context of model checking, the automaton B to be checked for emptiness is actually the product of a Kripke structure (representing the state space of the model under verification) with an automaton capturing the behaviors invalidating an LTL formula φ (the specification to check).

Theorem 1. *Let φ be an LTL formula, $A_{\neg\varphi}$ an automaton with n acceptance marks such that $\mathscr{L}(\neg\varphi) = A_{\neg\varphi}$, and K a Kripke structure. The following statements are equivalent:*

1. $\mathscr{L}(K) \subseteq \mathscr{L}(\varphi)$,
2. $\mathscr{L}(K) \cap \mathscr{L}(A_{\neg\varphi}) = \emptyset$,
3. $\mathscr{L}(K \otimes A_{\neg\varphi}) = \emptyset$,
4. $K \otimes A_{\neg\varphi}$ has no reachable, accepting cycle;
or in case $n \leq 1$ no reachable accepting elementary cycle,
5. $K \otimes A_{\neg\varphi}$ has no reachable, accepting SCC.

The emptiness checks we will present either look for accepting elementary cycles (when $n \leq 1$) or accepting SCCs. However an important point is that they search for those in the product $K \otimes A_{\neg\varphi}$. Because the Kripke structure K can be pretty large, a classical optimization is to generate both the Kripke structure K and the product $K \otimes A_{\neg\varphi}$ on the fly, as required by the needs of the emptiness-check procedure. Doing so avoids generating any part of K that would never be reached in the product, and it may also save a lot of time in case an accepting cycle is discovered early: the emptiness check can then exit immediately without exploring the rest of the product. For this on-the-fly construction to work, the emptiness check should only move *forward*, i.e., from a given state (s_1, s_2) of the product, one may only compute its successors, but not its predecessors. Originally, only the initial state (ι_1, ι_2) is known, and the emptiness check may explore the successors of this state, as well as the successors of any new state discovered this way. In such a setup, any cycle or SCC we discover is necessarily *reachable*.

12.2.7 Implicit Models and Automata

We have seen in Property 2 that the size of the Büchi automaton can be exponential in the size of the LTL formula, i.e., the number of symbols it contains. Not much has been said about the size of the model M. To expand on this, we first need to make some assumptions about its representation.

Definition 8. A *model* is a tuple $M = (D, \theta, \text{state-labels}, \text{next-state})$ where

- $D = V_1 \times \cdots \times V_k$ is the data of the model composed of k Boolean variables,
- $\theta \in D$ is the initial state,
- state-labels: $D \to \mathbf{2}^{\mathrm{AP}}$ is a state label function, and
- next-state: $D \to \mathbf{2}^{D}$ is a next-state function.

The data D of the model can be thought of as the values of all variables and program (thread) counters in some imperative language. The set D represents all *potential* states of the model. The next-state function provides an implicit encoding of all transitions in the system from a given state. It is typically an implementation of the system semantics of the individual program statements; for an example see [62].

The actual Kripke structure can be computed as an explicit representation of the data that the model represents implicitly.

Definition 9. The Kripke structure $K_M = (Q, \iota, \delta, \ell)$ of a model $M = (D, \theta, \text{state-labels}, \text{next-state})$ is defined as follows:

- $\iota = \theta$,
- Q is the smallest fixpoint of next-state that includes θ,
- $\delta = \big\{(s, d) \in D^2 \mid d \in \text{next-state}(s)\big\}$, and
- $\ell = \text{state-labels}$.

The introduction mentioned that the graph of the system (the Kripke structure of the model) is exponential in the number of components and variables. We can now be more exact. Let n be an upper bound on the data domains, i.e., $|V_i| \leq n$ $(0 \leq i \leq k)$.

Property 9. The number of states in the Kripke structure $K = (Q, \iota, \delta, \ell)$ is exponential in the number of variables in the model (k): $|Q| \in \mathrm{O}(n^k)$.

The implicit definition of the Kripke structure can be extended to the product automaton as well.

Definition 10 (Implicit product automaton). The *implicit product automaton* of a model $M = (D, \theta, \text{state-labels}, \text{next-state})$ and a TGBA $A = (Q, \iota, \delta, n, M)$ is the implicit TGBA $C = (Q', \iota', \text{next-product}, n, M')$ where:

- $\iota' = (\theta, \iota)$,
- $Q' = D \times Q$,

- $(x,(d_1,d_2)) \in \text{next-product}((s_1,s_2)) \iff (d_1) \in \text{next-state}(s_1) \wedge \text{state-labels}(s_1) = x \wedge (s_2,x,d_2) \in \delta$, and
- $M'\big(((s_1,s_2),x,(d_1,d_2))\big) = M\big((s_2,x,d_2)\big)$.

Definition 11. The TGBA $(Q'',\iota',\delta',n,M')$ *generated from* the implicit product automaton $(Q',\iota',\text{next-product},n,M')$ is defined by taking:

- Q'' is the smallest fixpoint of next-product that includes ι',
- $\delta' = \big\{(s,x,d) \in D^2 \mid (x,d) \in \text{next-product}(s)\big\}$.

Property 10. By definition, the product TGBA of M and A in Definition 11 is the same as $K_M \otimes A$ from Property 4.

Property 11. The number of states in the product structure $K_M \otimes A_{\neg\varphi} = (Q,\iota,\delta,n,M)$ of a model $M = (D,\theta,\text{state-labels},\text{next-state})$ and a TGBA $A_{\neg\varphi}$ can be exponential in the number of variables in the model ($|D| = l$, with data domains bounded by n) and in the formula φ: $|Q| \in \mathrm{O}(n^l \times \mathbf{2}^{|\varphi|})$.

The implicit definition helps us to avoid storing all transitions of the Kripke structure and its product, by recomputing them from the states. Moreover, entire parts of the Kripke structure might never have to be generated as they are suppressed by the synchronization of the product. The algorithms in the subsequent section will therefore use the implicit definition. While this definition prevents algorithms from doing backwards traversals (the inverse of next-state is not always computable), we will see that this is not required.

12.2.8 Simpler Subclasses

In 1990, Manna and Pnueli [76] presented a classification of temporal properties (i.e., languages expressed either as LTL or automata), into a hierarchy. Two subclasses are of particular interest in the context of model checking [25]: *guarantee* and *persistence* properties. The reason is that they can be represented by automata with additional constraints that simplify their emptiness checks.

Let us call an LTL guarantee (φ_G) and an LTL persistence (φ_P) any property that can be defined as an LTL formula using the following grammar, where $a \in AP$ is any atomic proposition. (φ_S and φ_R correspond to the dual classes of safety and recurrence.)

$$\begin{aligned}
\varphi_G &::= \bot \mid \top \mid a \mid \varphi_G \vee \varphi_G \mid \varphi_G \wedge \varphi_G \mid \mathsf{X}\varphi_G \mid \mathsf{F}\varphi_G \mid \varphi_G \,\mathsf{U}\, \varphi_G \mid \neg\varphi_S \\
\varphi_S &::= \bot \mid \top \mid a \mid \varphi_S \vee \varphi_S \mid \varphi_S \wedge \varphi_S \mid \mathsf{X}\varphi_S \mid \mathsf{G}\varphi_S \mid \varphi_S \,\mathsf{R}\, \varphi_S \mid \neg\varphi_G \\
\varphi_P &::= \varphi_S \mid \varphi_G \mid \varphi_P \vee \varphi_P \mid \varphi_P \wedge \varphi_P \mid \mathsf{X}\varphi_P \mid \mathsf{F}\varphi_P \mid \varphi_P \,\mathsf{U}\, \varphi_P \mid \varphi_P \,\mathsf{R}\, \varphi_S \mid \neg\varphi_R \\
\varphi_R &::= \varphi_S \mid \varphi_G \mid \varphi_R \vee \varphi_R \mid \varphi_R \wedge \varphi_R \mid \mathsf{X}\varphi_R \mid \mathsf{G}\varphi_R \mid \varphi_R \,\mathsf{R}\, \varphi_R \mid \varphi_R \,\mathsf{U}\, \varphi_G \mid \neg\varphi_P
\end{aligned}$$

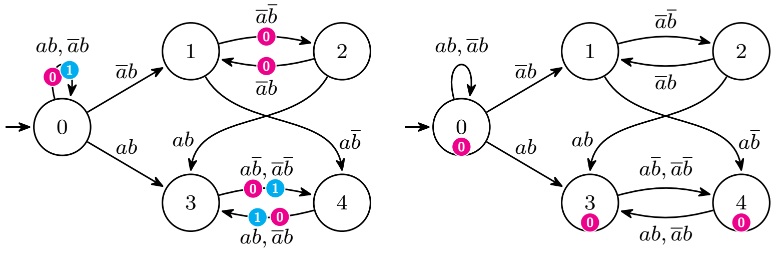


Fig. 12.5 A weak TGBA (left) and an equivalent weak SBA (right). Both have two accepting SCCs and one rejecting SCC. Inside each SCC, all transitions or states bear the same marks. Their language is that of the formula $(\mathsf{F}a \wedge \mathsf{G}((b \wedge \mathsf{X}\neg b) \vee (\neg b \wedge \mathsf{X}b)))\,\mathsf{R}\,b$, which is an LTL persistence

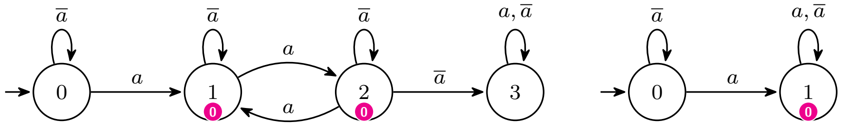


Fig. 12.6 Two terminal SBAs recognizing $\mathscr{L}(\mathsf{F}a)$. The left one was made artificially more complex to illustrate how any terminal automaton can be simplified by compacting all accepting SCCs into a single (and unique) state, and removing any SCCs that are only reachable via an accepting SCC

For instance, $\mathsf{GF}a$ is a recurrence formula (φ_R), $\mathsf{FG}b$ is a persistence formula (φ_P), but the conjunction of these two formulas $\mathsf{GF}a \wedge \mathsf{FG}b$ does not belong to any of the above classes.

LTL guarantee and LTL persistence formulas can be represented respectively by *terminal* and *weak* automata.

Definition 12 (Weak Automaton). A TGBA (or SGBA) is *weak* if in any of its SCCs all transitions (or states) have the same marks.

This definition implies that in each SCC of a weak automaton, either all cycles are accepting, or all cycles are rejecting. Because of that, any weak TGBA (Q, ι, δ, n, M) can be trivially converted into an equivalent SBA $(Q, \iota, \delta, 1, M')$, with the same transition structure, but defining M' by $M'(s) = [1]$ if there exists a transition $t \in \delta(s)$ such that $M(t) = [n]$ and t^s and t^d belong to the same SCC; or $M'(s) = \emptyset$ otherwise. Figure 12.5 illustrates this.

Weak automata can still express a large subclass of LTL properties. Many properties encountered in practice turn out to be weak or even simpler [11, 69].

An even simpler subclass of weak automata is terminal automata.

Definition 13 (Terminal Automaton). A TGBA (SGBA) (Q, ι, δ, n, M) is *terminal* if it is weak, and if any of its accepting SCCs is complete: that is, for any accepting SCC $C \subseteq Q$, any pair of states $s, d \in C$ within that SCC, and any assignment $x \in \mathbb{B}^{\mathrm{AP}}$, there exists $(s, x, d) \in \delta$.

The states that belong to accepting SCCs are called *terminal states.*

Note that because the accepting SCCs of terminal automata are complete, they will accept all suffixes. Therefore any terminal automaton can be simplified into an equivalent terminal automaton with a single terminal state looping over all possible assignments. Figure 12.6 illustrates this.

Property 12. From any LTL guarantee (φ_G on page 15) one can build an equivalent terminal automaton. Similarly, one can build a weak automaton equivalent to any LTL persistence (φ_P).

The subclass of LTL guarantees is simple enough that typical LTL translation algorithms [49, 6, 37] produce terminal automata naturally. A construction of weak automata from LTL persistence properties is given by Černá and Pelánek [25], and is implemented for instance in `ltl2tgba`.

The usefulness of terminal automata for model checking comes from the fact that to prove the existence of an accepting run, we only need to reach a terminal state. This fact also applies to the product with a Kripke structure, provided that the Kripke structure is known to be *deadlock-free* (Definition 2).

Property 13. Let $K = (Q_1, \iota_1, \delta_1, \ell)$ be a *deadlock-free* Kripke structure, and $A = (Q_2, \iota_2, \delta_2, n, M)$ a terminal automaton. $\mathscr{L}(A \otimes K) \neq \emptyset$ if and only if there exists a reachable state $(s_1, s_2) \in Q_1 \times Q_2$ where s_2 is a terminal state.

Indeed, the fact that K is dead lock-free implies that any prefix from ι_1 to s_1 can be continued into a lasso-shaped accepting run on K, and the fact that s_2 belongs to an accepting and complete SCC means that any suffix can be accepted from there. Therefore, upon reaching (s_1, s_2) it is clear that an accepting run can be found in $A \otimes K$.

In the subsequent section, we show that these simpler classes of automata also allow for simpler algorithms to solve the emptiness-check problem.

12.3 Basic Sequential LTL Model Checking Algorithms

The current section presents sequential algorithms for checking emptiness of Büchi automata. As discussed in the previous section, this problem can be solved in the case of $n \leq 1$ by showing that none of the elementary cycles are accepting. In the generalized case with $n \geq 2$, however, all cycles need to be considered according to Theorem 1. Therefore, we present a specialized algorithm called Nested Depth-First Search for the case where $n \leq 1$ and an SCC-based algorithm for the general case. We will show that the generality of the second algorithm comes at the cost of a slightly higher resource consumption.

We also saw that the automaton to check is the *product* between the property automaton $A_{\neg\varphi}$ and the Kripke structure K_M. Since this product can be large, a classical technique these algorithms employ is to compute this product *on the fly*. Before presenting the algorithms, we first discuss the on-the-fly technique and its advantages.

12.3.1 On-The-Fly Algorithms

While the automaton $A_{\neg\varphi}$ representing the specification is usually quite small (often fewer than 10 states), the automaton K_M can have billions of states, and the product of these two automata is a Cartesian product of their states in the worst case (i.e., $|K_M \otimes A_{\neg\varphi}| \leq |K_M| \otimes |A_{\neg\varphi}|$).

For efficiency reasons model checkers will therefore compute K_M and $K_M \otimes A_{\neg\varphi}$ *on the fly*, using the implicit definitions from Section 12.2.7. So instead of using the static definition of product transitions δ, we use its implicit counterpart next-product. This approach has various advantages:

- any part of K_M that does not synchronize with $A_{\neg\varphi}$ is not computed,
- we do not need to store the transitions of K_M and $K_M \otimes A_{\neg\varphi}$ since these can be recomputed when needed, and
- states can be deleted and recomputed, at the expense of re-explorations of the automaton, thus allowing for trading of computation time for memory use.[2]

The advantages are especially important when we recall that the number of states in the product automaton is exponential in both the property and the system (see Property 11). As memory is often a bottleneck for model checking, it would be disastrous to store those as well since there might be up to quadratically more transitions than states.

An important consequence is that these emptiness-check algorithms are only allowed to move forward in the automaton: from a state of A, one can compute the successors, but not the predecessors. This restriction comes from the fact that the actions of the original model might not be reversible (it might be intractable to compute the inverse of next-product). While respecting this constraint, the emptiness check needs to explore the product automaton to find information about cycles or SCCs.

[2] Various state space caching techniques have been invented that also ensure termination of the model checking algorithm [55, 89].

12.3.2 Depth-First Search

This exploration can be done using one of the two classical graph traversal algorithms: *breadth-first search* (BFS) or *depth-first search* (DFS). These algorithms iterate over vertices of a graph (or states of an automaton). The evolution of both DFS and BFS may be described as a process by which every state in the automaton is colored. At the beginning a state has no color and, at some point, it becomes "activated" and receives its color. In the general description of DFS below, we use $\bot$ for "no color" and $\top$ for "a color". These algorithms only differ by the order in which states are colored. In depth-first search, when choosing which state to explore next, children are favored over siblings. In contrast, in a breadth-first search siblings are favored over children. Even if both DFS and BFS have running time that is linear in the size of the product automaton (i.e., the number of states plus the number of transitions), most sequential emptiness checks are based on a DFS exploration since it can be used to detect cycles easily.

Algorithm 1: Depth-first search algorithm

```
1 function Setup ($A = (Q, \iota, \text{next-product}, n, M)$)
2     Dfs($A, \iota$)
3 function Dfs ($A = (Q, \iota, \text{next-product}, n, M), s$)
4     $s.color := \top$
5     forall $t \in \text{next-product}(s)$ do
6         if $t^d.color = \bot$ then
7             Dfs($A, t^d$)
```

Algorithm 1 presents a DFS exploration for an implicit automaton $A = (Q, \iota, \text{next-product}, n, M)$. Lines 1–2 only set up the exploration and launch the DFS exploration with the initial state ι of the automaton A. The main procedure (lines 3–7) maintains for each state a Boolean *color*, initially set to $\bot$, that keeps track of "activated" states. Every time a state is visited, its field *color* is set to $\top$ (line 4). At line 5 all successors of the currently visited state are processed: only new ones, i.e., with $color = \bot$, are recursively visited (line 7). The stack of recursive calls is also called the *DFS stack*. A state that is colored $\top$ and is on the stack is called *scheduled* or *stacked*. Once all its successors have been considered, it is popped off the stack, or *backtracked*.

A closer look at this algorithm shows that DFS exploration by its nature supports on-the-fly processing: only the initial state is used at the beginning (line 2) and the predecessors of a state are never computed (line 5).

The emptiness of a terminal automaton $A = (Q, \iota, \text{next-product}, n, M)$ (see Section 12.2.8) can easily be verified using the above DFS. All we have to do is to check whether $M(t) = [n]$ (for transition-based acceptance) or

$M(t^s) = [n]$ (for the state-based case) in the *for* loop. The check is so simple that it can be done by a BFS algorithm as well.

To detect elementary cycles of the automaton, the DFS algorithm has to be extended to keep track of the states on the stack. Algorithm 2 does this. It first marks the state s that is about to be explored *gray* at line 5. When backtracking over a state (removing it from the (program) stack), its color is set to *black* (line 11). When exploring the successor t^d of s at line 6, if t^d is in the DFS stack, a cycle has been found. Indeed, the states in the DFS stack between t^d and s form a path and t^d is a successor of s. Otherwise, if t^d is not on the DFS stack, no information about cycles can be inferred.

The algorithm exploits this to check the accepting condition in the weak case (Definition 12). Since in this case either all states on the cycle are accepting or none are, the following solution is correct. At line 2, the automaton is first converted into an equivalent state-based version. Then at line 7, the check for elementary cycles is performed by checking whether $t^d.color = gray$. If additionally the state t^d is accepting ($M(t^d) = [1]$), non-emptiness of the automaton is reported at line 8. We only need to check the accepting mark on t^d (or s), and not the marks of other states on the cycle, as all states in one SCC have the same mark by Definition 12 and consequently all states on the same cycle also carry the same mark.

Algorithm 2: Sequential emptiness check for weak TGBAs based on DFS

```
1  function Setup ($A = (Q, \iota, \text{next-product}, n, M)$)
2      Convert $A$ to an equivalent SBA $A'$ (e.g. Figure 12.5)
3      Dfs($A', \iota$)
4  function Dfs ($A = (Q, \iota, \text{next-product}, 1, M), s$)
5      $s.color := gray$
6      forall $t \in \text{next-product}(s)$ do
7          if $t^d.color = gray \land M(t^d) = [1]$ then
8              report non-empty
9          if $t^d.color = \bot$ then
10             Dfs($A, t^d$)
11     $s.color := black$
```

Edelkamp et al. [40] show how such simple algorithms can be used even in the case when only part of the automaton is weak or terminal. In Section 12.4.1, we discuss similar parallel variants.

Since the Büchi emptiness-check problem requires an inspection of all cycles to exclude accepting cycles, most algorithms rely on a DFS exploration (with some more elaborate cycle checks for general, non-weak TGBAs/SBAs as we will show in the subsequent section on Nested-DFS). These algorithms either use DFS directly to conclude emptiness by inspecting elementary cycles, exploiting Property 7, or decompose the automaton into SCCs, exploiting Property 8. *Nested-DFS* falls in the former category, while the SCC algorithm falls in the latter.

In contrast, a BFS exploration cannot easily detect cycles. Consequently, using BFS as exploration strategy requires a redesign of the LTL model checking algorithms, as we will illustrate in Section 12.5.

12.3.3 Nested-DFS

The Nested-DFS algorithm (NDFS) was originally proposed by Courcoubetis et al. [31] and relies on the detection of accepting elementary cycles reachable from the initial state. This algorithm focuses on SBA with $n \leq 1$ and runs in time linear with respect to the size of the graph. The algorithm accomplishes this by using DFS. Its use of DFS is however not as simple as we have seen in the previous section, because we cannot simply check the acceptance criterion on any state in the cycle as is sufficient in the case of weak automata.

NDFS uses a first DFS to detect *accepting states*, i.e., states of the automaton holding the unique acceptance mark. Traditionally this DFS is called *blue*-DFS since it colors in blue all the states encountered during the exploration. When an accepting state is about to be backtracked during this search, a second DFS is then invoked with the accepting state as a *seed*. This DFS colors all states in red and thus it is often called *red*-DFS. The goal of this second exploration is again to reach the seed state. If this state, which is accepting, can be reached itself, an accepting run is reported proving that the automaton has a non-empty language. Because the version in Algorithm 3 contains several improvements, we first discuss its details.

The BlueDfs function (lines 4–15) is similar to the DFS presented in Algorithm 1. Nonetheless some improvements have been added to transform it into an emptiness check. First of all, this algorithm uses two bits per state to keep track of the associated colors. Four colors are used:

- *white*: the initial color of a state. We assume that states are white when they are generated for the first time.
- *cyan*: the state is still in the DFS stack of the blue search.
- *blue*: all the direct successors of the state have been visited by the blue-DFS but not yet by a red one.
- *red*: states that have been considered in both the blue- and the red-DFS.

The BlueDfs function starts by coloring any new state in cyan (line 5). This color helps to detect accepting cycles directly inside the BlueDfs (lines 7 and 8): during this search, if the successor t^d of an accepting state s is cyan an accepting run exists since there is a path from d to s and vice versa. Similarly, if t^d is accepting and cyan, an accepting run exists. Otherwise, if t^d has not yet been visited (line 9) a recursive call is performed (line 10).

Algorithm 3: Nested Depth-first search algorithm

```
1  function NDFS ($A = (Q, \iota, \text{next-product}, n, M)$)
2      assert($n = 1$)
3      DFSBLUE($A, \iota$)
4  function DFSBLUE ($A = (Q, \iota, \text{next-product}, 1, M), s$)
5      $s.color := cyan$
6      forall $t \in \text{next-product}(s)$ do
7          if $t^d.color = cyan \wedge (M(t^s) = [1] \vee M(t^d) = [1])$ then
8              report non-empty
9          else if $t^d.color = white$ then
10             DFSBLUE($A, t^d$)
11     if $M(s) = [1]$ then
12         DFSRED($A, s$)
13         $s.color := red$
14     else
15         $s.color := blue$
16 function DFSRED ($A = (Q, \iota, \text{next-product}, 1, M), s$)
17     forall $t \in \text{next-product}(s)$ do
18         if $t^d.color = cyan$ then
19             report non-empty
20         else if $t^d.color = blue$ then
21             $s.color := red$
22             DFSRED($A, t^d$)
```

Two cases are of interest when all the successors of a state have been visited, i.e., just before backtracking it from the blue search. If the state is not accepting (line 15), its color becomes blue and the state is backtracked. Otherwise, the state is accepting (line 11) and the algorithm launches a nested exploration using the RedDfs function.

This function uses the accepting state as a *seed*, which is treated specially: it remains cyan during the red search and becomes red afterwards (line 13). This is required to limit the algorithm to four colors (which can be stored in two bits). The RedDfs function only looks for a state with the cyan color, i.e., a state that belongs to the DFS stack of the blue exploration. Because the stack of the blue search terminates in the seed, this condition is sufficient to demonstrate the reachability of a cycle over an accepting state. Therefore, if a cyan state is detected in the red search (line 18) then an accepting run exists and the automaton is reported to have a non-empty language (line 19).

Because the red search therefore never crosses the stack of the blue search, it will only explore blue states.

One can also note that all states visited by the RedDfs are marked red (line 21) and thus will be ignored by other (blue or red) explorations. This makes NDFS linear in the size of the input automaton (in terms of states and transitions). But why does the red search not have to reset its visited states like the inner search of the previous algorithm? It turns out that the DFS order of the blue search plays a crucial role here. Consider the case where the red search is started from a seed s and it encounters a red state. It can be shown that this state can never lead back to the cyan stack, because that would contradict the depth-first order of the blue search. An intuition for this property can be found in [48] and a detailed proof in [64].

Note that if the automaton has no accepting state the NDFS is optimal since states and transitions are visited only by the blue-DFS.

Many improvements of this algorithm have been proposed [59, 50, 40] to faster detect non-emptiness, reduce the size of accepting runs if they exist, or to reduce memory footprint. Algorithm 3, derived from the work of Schwoon and Esparza [87], presents a combination of all these optimizations.

12.3.4 Algorithms Based on SCC Decomposition

The algorithm presented in the previous section works only if the automaton to check is a non-generalized Büchi automaton. If the input automaton is a generalized one, the emptiness check of Tauriainen [94] can be used. This algorithm derives from the NDFS and repeats the inner DFS several times (at worst n times, with n the number of acceptance marks). The main drawback of this algorithm is that its complexity depends of the number of acceptance marks: this reduces all the benefits of using a generalized Büchi automaton.

Another idea to check for the emptiness of a generalized Büchi automaton is to degeneralize this automaton (as described by Property 1) before checking its emptiness. In this approach, the degeneralized automaton may have $n \cdot |Q|$ states, with $|Q|$ the number of states of the input automaton and n the number of acceptance marks. Once again, this approach is not optimal since it depends of the number of acceptance marks.

Another emptiness-check approach is to compute the accepting strongly connected components of the generalized Büchi automaton. SCC-based emptiness checks [32, 52, 33, 4, 48] are still based on a DFS exploration of the automaton; they do not require another nested DFS, have a linear time complexity and directly support TGBA. These emptiness checks are based on the classical SCC decomposition algorithm for directed graphs by Tarjan [90], which partitions the set of states according to the SCC equivalence classes. Each partition is then associated with the set of acceptance marks that appears inside the corresponding SCC to facilitate the emptiness check.

Intuitively, Tarjan's algorithm maintains a separate stack (apart from the search stack) of partial SCCs. Partial SCCs are enlarged when the DFS finds a cycle by adding its states to the secondary SCC stack. Each partial SCC is associated with a *potential root*, i.e., the state of the partial SCC that is the lowest on the stack. Thus, every time the partial SCC is enlarged, a new potential root may be selected. When the root is backtracked, the DFS order guarantees that the entire SCC was visited and is on the secondary stack. This is the moment when it is popped off the stack and the SCC can be reported even before the algorithm finishes traversing the entire graph (i.e., on the fly). To identify current roots the algorithm uses indices. Therefore, it uses slightly more memory per state than the NDFS algorithm, which requires only two bits per state.

We focus on a version of Tarjan's algorithm that maintains partial SCCs in a database, as it forms the basis of communicating partial SCCs in our parallel algorithm (see Section 12.4.3). It was developed by Purdom [80] even before Tarjan's algorithm, and later optimized by Munro [79]. Like Tarjan's algorithm it uses DFS, but this is not explicitly mentioned (Tarjan was the first to do so). In this algorithm, the secondary stack only stores roots as the partial SCC is kept in the database. We also add the ability to collapse cycles into partial SCCs immediately (as in Dijkstra [35, 47]).

The database with partial SCCs is implemented using a union-find data structure. As its name suggests, a union-find is a data structure that represents sets and provides efficient union and membership-check procedures. The union-find structure partitions a set E of elements and associates a unique representative (an element of E) with each partition. This structure offers the following methods on elements $x, y \in E$:

- $\textsc{makeset}(x)$: creates a new partition containing the element x if x is not already in the union-find.
- $\textsc{find}(x)$: returns null if x is not in the union-find, otherwise returns the actual representative of the partition containing x.
- $\textsc{sameset}(x, y)$: returns a Boolean indicating whether x and y are in the same partition.
- $\textsc{unite}(x, y)$: merges the partitions containing x and y.

With this structure, the set E of elements is partitioned into disjoint subsets $\{S_1, \ldots, S_m\}$ where m corresponds to the number of disjoint subsets. The underlying data structure of each subset S_i is typically a reverse arborescence (an in-tree), represented by a *parent* function $p(x) \in S_i$ for each $x \in S_i$. A unique representative y is appointed as the root of this in-tree. It is often designated with a self-pointer $p(y) = y$.

The parent function is usually implemented using an array of size $|E|$ that stores, for each element in $|E|$, the index of its parent in the tree. The array elements are initialized to $\perp$ representing the empty subset. The operation $\textsc{makeset}(x)$ then creates a singleton set consisting of its root $p(x) := x$. If two sets are merged with $\textsc{unite}(x, y)$, first the representativity of $r_x = \textsc{find}(x)$ and $r_y = \textsc{find}(y)$ is identified. Then one of them, e.g., r_y, is designated the new root by setting $p(r_x) := r_y$.

By compacting the paths in the in-tree, i.e., making leaves point directly to the root, the operations on the structure can all be solved in quasi-constant, amortized time [92]. Many variants on compaction schemes and unite strategies have been studied by Tarjan and van Leeuwen [93].

Algorithm 4: SCC-based emptiness check

```
1  Union-find of $\langle Q \uplus \{Dead\} \rangle$ : $uf$
2  Stack of $\langle q \in Q, a \in 2^{[n]}, ingoing \in 2^{[n]} \rangle$ : $roots$
3
4  function Setup $(A = (Q, \iota, \text{next-product}, n, M))$
5      $uf$.makeSet($Dead$)
6      SccBased($A, \iota, \emptyset$)
7  function SccBased $(A = (Q, \iota, \text{next-product}, n, M), s, acc)$
8      $uf$.makeSet($s$)
9      $roots$.push($\langle s, \emptyset, acc \rangle$)
10     forall $t \in \text{next-product}(s)$ do
11         if $uf$.sameSet($t^d$, $Dead$) then
12             continue
13         else if $uf$.find($t^d$) $= null$ then
14             SccBased($A, t^d, M(t)$)
15         else
16             $roots$.top().$a \leftarrow roots$.top().$a \cup M(t)$
17             while $\neg uf$.sameSet($t^d, s$) do
18                 $\langle r, a, i \rangle \leftarrow roots$.pop()
19                 $roots$.top().$a \leftarrow roots$.top().$a \cup i \cup a$
20                 $uf$.unite($r, roots$.top().$q$)
21             if $roots$.top().$a = [n]$ then
22                 report non-empty
23     if $roots$.top().$q = s$ then
24         $roots$.pop()
25         $uf$.unite($s$, $Dead$)
```

Algorithm 4 presents the emptiness check [83] for TGBA. Two global variables are used:

1. The union-find uf (line 1), which stores the various partitions corresponding to the SCCs discovered so far by the exploration. This structure maintains a special partition *Dead*, which holds all states of already completed SCCs (without accepting run), i.e., all states that cannot be part of an accepting run.
2. The roots stack *roots* (line 2) that contains tuples composed of: q the potential root, a the set of acceptance marks (visited so far) associated with the SCC containing q, and a special field *ingoing*. This special field keeps track of the acceptance marks held by the ingoing transition. This information must be kept since it is not directly available on TGBAs.

Lines 4 to 6 only set up the union-find with the special partition *Dead*, and then call the recursive exploration through the SccBased function. This function takes three parameters: the automaton to check, the state to explore, and the acceptance mark held by the ingoing transition.

Lines 8 and 9 respectively insert the state into the union-find and the roots stack. Lines 10 to 22 process all the successors of the current state s. If the destination t^d of a transition is already Dead (lines 11–12) then the transition is just skipped since it cannot lead to an accepting run. If the destination has not yet been visited (lines 13–14) the function is called recursively. Finally, the destination can be a part of an SCC (trivial or not) that is not yet marked *Dead*. In this case, a cycle has been found and partial SCCs stored in the roots stack (lines 16–20) must be merged. During this merge the acceptance marks in the SCC are also merged (line 19). When all partial SCCs have been merged, an accepting run exists iff the field a of the top of the roots stack contains all acceptance marks. Note that this test could also be done during the merge.

Finally, when the root of an SCC is about to be backtracked, all states belonging to this SCC must be marked *Dead*. Line 25 performs this operation in quasi-constant time, by virtue of the union-find data structure.

12.4 Multi-core, DFS-Based Solutions

12.4.1 Terminal and Weak Acceptance

In Section 12.3, we saw that the simplest classes of Büchi automata often allow for simpler and more efficient algorithms. Here we show that checking emptiness of weak and terminal automata can be done using a parallel version of DFS that preserves enough of the depth-first order to still be able to find all elementary cycles. First, we show how a simple parallel search can detect emptiness of terminal automata, as it illustrates nicely what low-level ingredients are required for shared-memory parallel algorithms.

Terminal acceptance

Algorithm 5 shows a parallel search algorithm with a shared state set. To simplify the acceptance condition, the algorithm first converts the terminal automaton, which is by extension also a weak automaton, to an equivalent SBA A' at line 4. Then it schedules the initial state in the stack or the queue of the first worker *Queues*[0]. The first worker will start exploring from this state and generate new states, as we will see later, while a load balancer will take care that work arrives in the queues of the other workers. When the initializations are completed, the algorithm launches the actual search procedure in parallel at line 7. At the first encounter of an accepting state the algorithm terminates at line 15, just like the sequential algorithm for terminal acceptance discussed in Section 12.3.2.

Algorithm 5: A parallel search algorithm for checking the emptiness of terminal automata

```
1  global $Queues[P]$
2  global $StateSet$
3  function PAR-TERMINAL-CHECK $(A = (Q, \iota, \text{next-product}, n, M), P)$
4      Convert $A$ into an equivalent SBA $A'$ (e.g. Figure 12.5).
5      $Queues[0] := \{\iota\}$
6      $StateSet := \emptyset$
7      $\text{SEARCH}_1(A') \mid\mid \ldots \mid\mid \text{SEARCH}_P(A')$
8      report no-cycle
9  function SEARCH$_p$ $(A = (Q, \iota, \text{next-product}, 1, M))$
10     while $load\text{-}balance(Queues[p])$ do
11         $s := Queues[\text{p}].dequeue()$
12         if $StateSet.find\text{-}or\text{-}put(s)$ then
13             forall $t \in \text{next-product}(s)$ do
14                 if $M(t^d) = [1]$ then
15                     report cycle and terminate
16                 $Queues[p].queue(t^d)$
```

Each worker perpetually calls the load balancer at line 10. When its queue is non-empty ($Q[p] \neq \emptyset$), the *load-balance* function will merely return true. When a worker has run out of work ($Q[p] = \emptyset$), however, the function takes some work from the queue of another thread and adds it to the local queue $Q[p]$. Only when the load balancer detects termination, using a specialized termination detection algorithm [85], will the load balancer return false, allowing the worker thread to exit the SEARCH function.

The use of a load balancer has the advantage that no communication occurs while workers still have work locally available (their queue is non empty). Only in the extreme cases when a worker is without work, e.g., right after initialization and when most of the state space has been processed, will the algorithm experience overhead from additional synchronization. Specialized concurrent "deque" data structures allow the load balancer to be particularly efficient [19].

For the rest, the parallel search function operates as expected: A state is taken from the local queue at line 11, its successors are considered at line 13, and when a new state is encountered it is added to the local queue at line 16. The worker thus traverses the state space more or less independently, with one exception: visited states are entered into a shared set *StateSet*. To atomically add states, this set implementation has a *find-or-put* operation, which at the same time checks whether a state s is already contained in the set, and when this is not the case, adds it to the set. It can be used to "grab" new states and thus exclusively assign them to the worker that encounters a state first.

The state set can be implemented efficiently as a concurrent hash table or tree table data structure [71, 68]. Because the set of visited states accounts for almost all memory use of the algorithm (recall from the previous section that transitions do not need to be stored), and because workers diverge into different parts of the (huge) state space, most lookups in the table do not collide, i.e., they access different parts of the table. This is another efficient aspect of the algorithm; it exploits the random memory characteristic of model checking algorithms (as also discussed in the introduction) to increase parallelism.

In the sequential case, the algorithm yields a strict DFS order when implementing *Queues* as a stack, and a strict BFS order when implementing *Queues* as a fifo-queue. This parallel algorithm variant however violates a strict order as soon as workers start encountering the same states. Because only one of them will win the race in the *find-or-put* call, the others are forced to violate the order. For this reason, the algorithm might just as well immediately try to "grab" each generated state t^d inside the for loop by moving line 12 right before line 16 (the state set should be initialized to $\{\iota\}$). While this causes a more abnormal search order, it limits all duplication of states on local stacks.

Various researchers have found ways to approach BFS more precisely in parallel algorithms, while also limiting communication by introducing separate queues [2, 58]. A more precise order can have practical benefits, e.g., it allows the model checker to find the shortest counterexample, but also mitigates the on-the-fly behavior of the procedure. It is unknown yet whether (non-lexicographic) DFS can be preserved efficiently as well (recall from the introduction that lexicographic DFS, with fixed transition ordering, likely is not parallelizable according to theory). Nonetheless, we now show that with a simple parallel algorithm, we can preserve enough of the DFS order to find all elementary cycles, which is sufficient to tackle the LTL model checking problem as the following sections show.

Weak acceptance

Emptiness of weak automata is a little harder to compute than for terminal automata because the algorithm still needs to inspect all elementary cycles. In Section 12.3.2, we showed how DFS can solve it sequentially. Algorithm 6 does the same in parallel. Again, to simplify the acceptance condition, the algorithm first converts the terminal automaton to an equivalent SBA A' at line 2. Then, the algorithm launches the actual search procedure in parallel at line 3. All workers start searching from the same initial state.

Algorithm 6: A parallel DFS algorithm for checking emptiness of weak automata

```
function PAR-WEAK-CHECK ($A = (Q, \iota, \text{next-product}, n, M), P$)
    Convert $A$ to an equivalent SBA $A'$ (e.g. Figure 12.5)
    PAR-DFS$_1(A', \iota)$ || ... || PAR-DFS$_P(A', \iota)$
    report no-cycle
function PAR-DFS$_p$ ($A = (Q, \iota, \text{next-product}, 1, M), s$)
    $s.gray[p] := true$
    forall $t \in$ RANDOMIZE(next-product($s$)) do
        if $t^d.gray[p] \wedge M(t^d) = [1]$ then
            report cycle and terminate
        if $\neg t^d.gray[p] \wedge \neg t^d.black$ then
            PAR-DFS$_p(t^d)$
    $s.black := true$
    $s.gray[p] := false$
```

The search procedure resembles the sequential DFS procedure of Algorithm 2, with the exception that the stack states are now colored *gray* locally. This means that workers' stacks might overlap while searching through the state space. When backtracked, however, the states are colored globally *black*, pruning the search space for other workers. This is where the speedup of the parallel algorithm comes from. To obtain the best performance, the search order of each parallel worker should be randomized, so that workers are guided into different parts of the state space [65]. Although redundant due to the set inclusion, we nonetheless emphasize this with the RANDOMIZE function.

To detect cycles, the algorithm uses the same stack-based check as its sequential counterpart. It will not miss any cycles because of the parallel search for the following reasons:

- It is possible to show that all black states always have black or gray states as successors (gray for some worker).
- When a worker p ignores a state t^d for being black, and that state actually has a path to its gray stack, then by induction on the cycle, it can be shown that there is some other worker in a similar situation or able to find a path back to its stack.
- Because there are a finite number of workers, one will eventually find the cycle.

A full proof of correctness can be found in Laarman and Faragó [69].

Because the use of DFS, the weak emptiness check algorithm looks simpler than Algorithm 5. Indeed, it does not require a load-balancer, because work distribution is achieved by letting stacks (partly) overlap. While it may be the case that workers exclude each other from parts of the state space, there are easy ways to remedy that [69]. Because the lack of a load-balancer, the stack can be completely local (here it is maintained as part of the program stack). However, it is not the case that the algorithm does without a global state set. The set is hidden behind the color variables and implicitly accessed when these are referenced in the algorithm. Therefore, an efficient concurrent hash table or tree data structure is again crucial for its performance.

To detect non-progress properties, another subset of LTL, Laarman et al. [69] introduce DFS-FIFO, an algorithm that utilizes a similar parallel DFS. It can be used for checking emptiness of weak automata as well and improves the parallel scalability by combining the search with a highly scalable BFS. A similar approach was taken for parallel checking of weak LTL properties on timed automata in [34]. The parallel DFS approach has the additional benefit that it combines well with state space reduction techniques, as these can implemented with the same on-the-fly algorithm [70].

12.4.2 CNDFS

Two algorithms were presented simultaneously (LNDFS by Laarman et al. [66] and ENDFS by Evangelista et al. [42]) that adapted the Nested-DFS (NDFS) algorithm to multi-core architectures. Both share the principle of launching multiple instances of NDFS that synchronize themselves to avoid useless state revisits, just like the algorithm for checking emptiness of weak automata discussed in the previous section. Although they are heuristic algorithms in the sense that, in the worst case, they reduce to spawning multiple unsynchronized instances of NDFS, the experiments reported by Laarman et al. [66, 65] show good practical speedups.

They were then combined and improved in the CNDFS algorithm by Evangelista et al. [43]. This algorithm is both much simpler and uses less memory, making it more compatible with exact compression techniques such as tree compression [68] that can compress large states down to two integers.

CNDFS is presented in Alg. 7 for P threads. It is based on the principle of SWARM worker threads (indicated by subscript p here), sharing information via colors stored in the visited states: here *blue* and *red*. After randomly visiting all successors (lines 13–15), a state is marked blue at line 16 (meaning "globally visited"), causing the (other) blue-DFS workers to lose the strict postorder property.

If the state s is accepting, as in the sequential NDFS algorithm, a red-DFS is launched at line 19 to find a cycle. At this point, state s is called "the seed." All states visited by DFSRED$_p$ are collected in $\mathcal{R}_p$. If no cycle is found in the red-DFS, none exists for the seed. Still, because the red-DFS was not necessarily called in postorder, other (non-seed, non-red) accepting states may be encountered about which we know nothing, except the fact that they are out of order and reachable from the seed. These are handled after completion of the red-DFS at line 20 by simply waiting for them to become red.

In this scenario there is always another worker that can color such a state red. The intuition behind this is that there has to be another worker to cause the out-of-order red search in the first place (by coloring blue) and, in the second place, this worker can continue its execution because cyclic waiting configurations can only happen for accepting cycles. These accepting cycles would however be encountered first, causing termination and a cycle report (line 8). After completion of the waiting procedure, CNDFS marks all states in $\mathcal{R}_p$ globally red, pruning other red-DFSs.

Algorithm 7: CNDFS, a multi-core algorithm for LTL model checking

```
1  function cndfs ($\iota, P$)
2      $\text{dfsBlue}_1(\iota) \mid\mid \ldots \mid\mid \text{dfsBlue}_P(\iota)$
3      return no-cycle
4  function $\text{dfsRed}_p(A = (Q, \iota, \text{next-product}, n, M), s)$
5      $\mathcal{R}_p := \mathcal{R}_p \cup \{s\}$
6      forall $t \in \text{randomize}(\text{next-product}(s))$ do
7          if $t^d.cyan[p]$ then
8              return cycle and terminate
9          if $t^d \notin \mathcal{R}_p \land \neg t^d.red$ then
10             $\text{dfsRed}_p(A, t^d)$
11 function $\text{dfsBlue}_p(A = (Q, \iota, \text{next-product}, n, M), s)$
12     $s.cyan[p] := true$
13     forall $t \in \text{randomize}(\text{next-product}(s))$ do
14         if $\neg t^d.cyan[p] \land \neg t^d.blue$ then
15             $\text{dfsBlue}_p(A, t^d)$
16     $s.blue := true$
17     if $M(s) \neq \emptyset$ then
18         $\mathcal{R}_p := \emptyset$
19         $\text{dfsRed}_p(A, s)$
20         await $\forall s' \in \mathcal{R}_p$ s.t. $M(s') \neq \emptyset : s \neq s' \Rightarrow s'.red$
21         forall $s' \in \mathcal{R}_p$ do
22             $s'.red := true$
23     $s.cyan[p] := false$
```

An efficient parallelization of the blue-DFS is absolutely essential for scalability, since the number of blue states (all reachable states) typically exceeds the number of red states (visited by the red-DFS). Since it was impossible to color both blue and red while backtracking from the respective DFS procedures, CNDFS uses an intermediate solution, using a wait statement as a compromise, leaving enough parallelism to maintain scalability.

CNDFS only uses $P+2$ bits per state plus the sizes of $\mathcal{R}$. In the theoretical worst case (an accepting initial state), each worker $p \in [P]$ could collect all states in $\mathcal{R}_p$. According to extensive experiments, the set rarely contains more than one state and never more than thousands, which is still negligible compared to $|Q|$.

12.4.3 Multi-core/DFS-Based SCC Decomposition

To handle emptiness checking of TGBAs, a parallel SCC-based algorithm is required as Theorem 1 indicates. Traditional parallel SCC algorithms [86, 46, 13, 98, 60, 88] are BFS-based implementations of divide-and-conquer approaches, which are not on the fly [18]. Also, these algorithms often exhibit an $n \times \log(n)$ or quadratic-time worst-case complexity. We therefore rely on DFS to detect SCCs in parallel since DFS-based SCC detection can be both on the fly and linear time. The main difficulty here, like in the previous section, is that a sufficient amount of the DFS order must be preserved for correctly detecting cycles.

We first briefly discuss a fully synchronized approach and show how bottlenecks impose limitations on the algorithm's performance. Then we present a random search/swarmed approach that performs linearly and show how this technique scales for multiple workers.

Fully synchronized parallel SCC algorithm

The general idea of the fully synchronized algorithm [74] is to have multiple non-overlapping search instances. Every reachable state is visited by exactly one worker, who globally takes ownership of the state. Searches are spawned from unvisited successor states. Upon encountering a state taken by a different worker, the search suspends until the state is marked as being completely explored. Otherwise, the search proceeds similarly to Tarjan's algorithm [91].

A cycle of suspended searches can occur as a consequence. In case no further actions are taken, the algorithm may never finish. A map of suspended searches is used to detect such cycles. If a worker suspends a search and detects a cycle of suspended searches, it transfers all relevant states from the suspended searches to one search and proceeds normally. For example, suppose that a worker visits edge $v \rightarrow w$ and detects that w is part of a different search. Before suspending, it checks whether the path $w \rightarrow^* v$ can be found by traversing states from the suspended searches. If so, a cycle is detected, which should be resolved by the current worker.

Maintaining the suspended map and resolving cycles of suspended searches is a costly process. The sequential linear-time performance of Tarjan's algorithm reduces to a quadratic worst-case performance in the synchronized variant. For the practical performance of the algorithm, two important cases can be distinguished: graphs containing relatively *large* SCC sizes ($|C| \sim |Q|$), often consisting of many interconnections; and *small* SCCs, consisting of only a few states ($|C| \sim 1$). The synchronized algorithm exhibits good scalability for graphs containing only small SCCs, since the different searches do not tend to interfere with each other. For graphs with large SCCs, a fully synchronizing algorithm can pay a large performance penalty if the worst-case time complexity is attained due to the wait-cycle checks. On the other hand, this algorithm totally avoids any redundant explorations as searches never overlap. Hence, while in appearance similar to the multi-core NDFS approaches discussed in the previous subsection, the fully synchronous algorithm has characteristics similar to the BFS-based algorithms that will be discussed in Section 12.5.

Swarmed parallel SCC algorithm

A different approach is to detect SCCs in a *swarmed* fashion, similarly to Cndfs (Section 12.4.2). The general idea of the algorithm is to spawn multiple instances of a sequential DFS algorithm and communicate the fully explored SCCs in a shared data structure [84]. An SCC is considered to be fully explored when all its successors (direct or indirect) have been explored. As a consequence, an instance of the algorithm can ignore all states belonging to a fully explored SCC. Thus, communicating fully explored SCCs allows us to prune other DFSs since an instance will never traverse a state that belongs to a fully explored SCC.

In this approach, two instances can still visit the same SCC (in a swarmed fashion) until one of the instances detects that it has been fully explored. If the SCC contains an accepting run, we want to be able to speed up its discovery. The multiple instances can then share the acceptance marks discovered so far for each (partial) SCC. This information helps us to find whether an accepting run exists. Suppose that we have two instances of a classical SCC-based algorithm running on the example of Figure 12.7 without sharing acceptance marks. Neither of these instances can detect an accepting cycle before $\delta_0, \delta_1, \delta_2$, and δ_3 have all been visited. Let us now suppose that they share acceptance marks and that the first instance i_0 has visited δ_1 and δ_2 while the other instance i_1 has visited δ_0. When instance i_1 discovers the transition δ_3 it also discovers that s_0 and s_1 are in the same SCC. In this case, since i_0 and i_1 share information about acceptance marks, they can detect the existence of an accepting cycle.

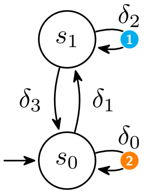


Fig. 12.7 Sharing acceptance marks

In the sequential SCC-based emptiness check (Algorithm 4) the information about fully explored SCCs is already stored inside a union-find data structure with a dedicated partition *Dead*. *Lock-free* versions of the union-find structure exist [5]. A simple implementation of this structure is presented in Algorithm 8. As mentioned Section in 12.3.4 each element stored by the union-find maintains a field *parent* which represents a forest of reverse arborescences. In a parallel setting this field must not be updated concurrently by two threads. This can be done using a *compare-and-swap* (CAS) operation (line 13 and 15). This operation is an atomic instruction used in multithreading to achieve synchronization: CAS(m, v_1, v_2) compares the contents of a memory location m to a given value v_1 and, only if they are the same, modifies the contents of that memory location to a given new value v_2. The CAS operation returns true if the modification was successful and false otherwise. A closer look to Algorithm 8 shows that this structure is only *lock-free* and not *wait-free* because of the spin-wait loops of lines 8 and 19. The rest of this union-find remains similar to the sequential version apart from the use of atomic operations.

This union-find can then be shared among the multiple instances to communicate fully explored SCCs. This structure can also be extended to store, for each partition, a set of acceptance marks. This modification slightly impacts the interface of union-find:

- When makeSet(e) effectively creates a partition for e (because it did not exist before), the associated acceptance set is $\emptyset$.
- The unite function takes an extra argument representing the set of acceptance marks that occur in the (partial) SCC. During this operation, the union-find must propagate the acceptance set to the representative of the partition. This is costless since this representative is already computed by the find function. Also note that for implementation details, a union with the partition containing *Dead* always returns $\emptyset$.

This swarmed emptiness check is presented in Algorithm 9 and mostly relies on the sequential SCC-based emptiness check presented in Algorithm 4. It performs a DFS, maintains a roots stack, and uses a union-find to store partitions representing partial SCC and *Dead* states. Nonetheless, some minor changes have been made:

- Only the union-find is shared among the threads. The roots stack is local to each instance.
- Each instance p now maintains a local integer $counter_p$ (line 3). This integer is only incremented (line 10) so it can be used to (locally) order states that have been visited.
- For an instance p, each state s is associated with a *live number*, i.e., an integer accessible via $s.livenum_p$. This live number is given according to $counter_p$ the first time the state is visited by the thread p (line 11).
- Line 21 has been changed since sameSet cannot be used to pop the roots stack until the new root is discovered. Indeed, since the union-find is shared among all threads, no assumptions about its internal state can be made.

Algorithm 8: Concurrent union-find data structure

```
function FIND ($a$)
    if $a.parent \neq a$ then
        $a.parent := \text{FIND}(a.parent)$
    return $a.parent$
function UNITE ($a, b$)
    $x := a$
    $y := b$
    while $true$ do
        $x := \text{FIND}(x)$
        $y := \text{FIND}(y)$
        if $x = y$ then return
        else if $x < y$ then
            if $\text{CAS}(x.parent, x, y)$ then return
        else
            if $\text{CAS}(y.parent, y, x)$ then return
function sameSet ($a, b$)
    $x := a$
    $y := b$
    while TRUE do
        $x := \text{FIND}(x)$
        $y := \text{FIND}(y)$
        if $x = y$ then return TRUE
        else if $x.parent = x$ then return FALSE
```

It is worth noting that the union-find structure collects the acceptance marks that are discovered by all threads. Thus, at line 23 the algorithm uses the global *uf* structure to detect which acceptance marks have been found, by any worker, in the partial SCC. This helps speed up reporting the existence of an accepting run. Nonetheless, if an SCC is not accepting, its states cannot be marked *Dead* before a thread has visited all the states and all the transitions of this SCC. This is a serious drawback of this algorithm when the automaton to check is composed of a single large SCC: in this case, the expected speedup is null. The next algorithm solves this problem.

Improved parallel swarmed SCC algorithm

The key aspects of the improved algorithm are to communicate partially found SCCs and globally track the remaining work left for each SCC. The SCC algorithm is presented in [18] and is applied to LTL model checking in [17]. It is presented in Algorithm 10 and differs slightly from Algorithm 9.

The local *counter* and *livenum* have been replaced by globally tracking worker IDs in the union-find structure. This *worker set*, $w \in 2^P$, is a bit-

Algorithm 9: Swarmed SCC-based algorithm

```
1  Shared Union-find of $\langle Q \cup \{Dead\}, a \in 2^{[n]} \rangle$ : $uf$
2  Local Stack of $\langle q \in Q, ingoing \in 2^{[n]} \rangle$ : $roots_p$
3  Local Integer $counter_p$
4
5  function SETUP $(A = (Q, \iota, \text{next-product}, n, M))$
6      $uf$.MAKESET$(Dead)$
7      $counter_1 \leftarrow 0;\ \ldots;\ counter_P \leftarrow 0$
8      SWARMEDSCCBASED$_1(\iota, \emptyset)$ || ... || SWARMEDSCCBASED$_P(\iota, \emptyset)$
9  function SWARMEDSCCBASED$_p$ $(A = (Q, \iota, \text{next-product}, n, M), s, acc)$
10     $counter_p := counter_p + 1$
11     $s.livenum_p := counter_p$
12     $uf$.MAKESET$(s)$
13     $roots_p$.PUSH$(\langle s, acc \rangle)$
14     forall $t \in$ RANDOMIZE(next-product$(s)$) do
15         if $uf$.SAMESET$(t^d, Dead)$ then
16             continue
17         else if $uf$.FIND$(t^d) = null$ then
18             SWARMEDSCCBASED$_p(A, t^d, M(t))$
19         else
20             $uf$.FIND$(s).a := uf$.FIND$(s).a \cup M(t)$
21             while $t^d.livenum_p < roots_p$.TOP$().q.livenum_p$ do
22                 $\langle r, i \rangle := roots_p$.POP$()$
23                 $uf$.UNITE$(r, roots_p$.TOP$().q, i)$
24             if $uf$.FIND$(s).a = [n]$ then
25                 report non-empty
26     if $roots_p$.TOP$().q = s$ then
27         $roots_p$.POP$()$
28         $uf$.UNITE$(s, Dead, \emptyset)$
```

set that tracks which worker threads are *active* in the current SCC. The MAKESET(p, s) is extended to set the bit for worker p in the partial SCC of s, which is tracked in the representative of the set. This worker set is used in line 14 to detect a cycle. Note that if worker p has visited some state s in a partial SCC, every state of this partial SCC is considered to have been visited before. This is valid since there is a path from every other state in the SCC to s. Also note that multiple workers aid each other by concurrently adding more states to the set, thus increasing the number of states that have been "visited before."

In order to collaborate in detecting when an SCC has been fully explored, the union-find structure has been further extended to track a list of *Busy* states in each partial SCC. The idea is to initially keep a global list consisting of every state in the SCC. Then, after concluding that no new knowledge can be obtained from a state, it gets removed from the list and another state is

Algorithm 10: UFSCC Algorithm: Improved swarmed SCC algorithm

```
1  **Shared Union-find** of $\langle Q \cup \{Dead\}, a \in 2^{[n]}, w \in 2^{P}, list \in 2^{Q} \rangle$ : $uf$
2  **Local Stack** of $\langle q \in Q, ingoing \in 2^{[n]} \rangle$ : $roots_p$
3
4  **function** SETUP $(A = (Q, \iota, \text{next-product}, n, M))$
5      $uf$.MAKESET$(Dead)$
6      IMPROVEDSCC$_1(\iota, \emptyset)$ || ... || IMPROVEDSCC$_P(\iota, \emptyset)$
7  **function** IMPROVEDSCC$_p$ $(A = (Q, \iota, \text{next-product}, n, M), s, acc)$
8      $uf$.MAKESET$(p, s)$
9      $roots_p$.PUSH$(\langle s, acc \rangle)$
10     **while** $s' \in uf$.PICKFROMLIST$(s)$ **do**
11         **forall** $t \in$ RANDOMIZE(next-product$(s')$) **do**
12             **if** $uf$.SAMESET$(t^d, Dead)$ **then**
13                 **continue**
14             **else if** $p \notin uf$.FIND$(t^d).w$ **then**
15                 IMPROVEDSCC$_p(A, t^d, M(t))$
16             **else**
17                 $uf$.FIND$(s).a := uf$.FIND$(s).a \cup M(t)$
18                 **while** $\neg$SAMESET$(s, t^d)$ **do**
19                     $\langle r, i \rangle := roots_p$.POP()
20                     UNITE$(r, roots_p$.TOP$().q, i)$
21                 **if** $uf$.FIND$(s).a = [n]$ **then**
22                     **report non-empty**
23         $uf$.REMOVEFROMLIST$(s')$
24     $uf$.UNITE$(s, Dead, \emptyset)$
25     **if** $roots_p$.TOP$() = s$ **then**
26         $roots_p$.POP()
```

chosen. In the algorithm this is shown in lines 10 and 23. When all successors of state s' have been handled (lines 11–22) we can conclude that for every successor d of s' we either have: (1) d is part of a *Dead* SCC, or (2) d is part of the same SCC as s'. In the latter case, d has been added to the list of *Busy* states and therefore s' can be removed from the list. Multiple workers pick states from the list, explore them, and correspondingly remove them from the list to cooperatively reduce the number of states in the list. Once the list is empty (exit condition for line 10), every state of the SCC has been fully explored and the SCC can be marked *Dead*.

In the implementation, the union-find structure is extended such that every state contains a worker set of size $|P|$, which is maintained (similarly to the acceptance set) in the UNITE procedure. Every state in the structure also contains a list-next pointer such that a cyclic list is formed of all states in the partial SCC. See Figure 12.8 for an illustration. Combining two lists in the UNITE procedure is then realized by swapping two list pointers with a fine-grained lock to form a single list containing all states. A Boolean flag is

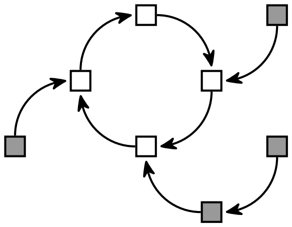


Fig. 12.8 Cyclic list of *Busy* states. White nodes are *Busy* and gray nodes have been removed from the list

used to mark a state to be removed from the list. Workers then traverse the list to find *Busy* states and update the next pointers such that the *removed* states are detached from the list.

12.5 Distributed, BFS-Based Solutions

In shared-memory, parallel algorithms can exploit relatively fast accesses to concurrent data structures to dynamically distribute the search procedure over the processor cores, as achieved in the previous section through the use of a shared state set. In the distributed setting, such synchronous communication would be too costly. To solve this problem, distributed algorithms statically partition the states over the workers, using so called *hash-based partitioning*. Under this scheme, every state of the graph to be stored is assigned to a single workstation that is responsible for its storage. The function to assign an owner of a state is referred to as the *partition* or *owner* function.

This section discusses two algorithms suitable for distributed computation. Because the static partitioning works best in combination with the highly scalable breadth-first search, the emptiness-check problem is first rephrased so that it can be solved by an iterative approach. In the worst case, each iteration represents one pass over the state space, but can be implemented with BFS. Nonetheless, for many inputs the time complexity of this approach is still optimal and we demonstrate that the emptiness check can even be made partially on the fly. At the end of this section, we show how this approach compares to the DFS approaches in the previous section.

12.5.1 One-Way-Catch-Them-Young

The emptiness-check algorithm discussed in this section is built on top of a procedure for topological sorting. It relies on the fact that vertices of a directed graph may be topologically sorted if and only if the graph is acyclic.

The topological sort procedure may be effectively adapted for parallel processing without any increase in the theoretical time complexity. While topological sort can directly detect the presence of a cycle in a directed graph, it cannot distinguish between accepting and non-accepting cycles. Therefore, it must be accompanied by another technique in order to be used as a Büchi automata emptiness check. One of the options to achieve accepting-cycle detection is to combine the topological sort procedure with a forward reachability analysis that eliminates states not reachable from accepting states. The algorithm relying on this combination is referred to as the OWCTY algorithm (One-Way-Catch-Them-Young) [44, 26].

The idea of the OWCTY algorithm is to remove leading rejecting SCCs (SCCs without accepting states) from the graph of the product Büchi automaton, then use the topological sort procedure to remove leading accepting states that do not lie on a cycle. This process is iterated until a fixpoint is reached. When the remaining graph is empty, it contains no accepting cycle. When the remaining graph is non-empty, the presence of an accepting cycle in the graph is guaranteed.

The OWCTY algorithm therefore uses two removal procedures, ELIM-NO-ACCEPTING and ELIM-NO-PREDECESSORS, which alternate. See Algorithm 11 for details. ELIM-NO-ACCEPTING is a procedure that computes all states that are reachable from an accepting state in the graph and removes the rest. Efficiently, this procedure removes all leading SCCs that contain no accepting states at all. Obviously, these SCCs must be rejecting. After that, all leading SCCs in the rest of the graph contain an accepting state; however, they might all be trivial SCCs (contain no edges). To detect whether there is a non-trivial leading SCC with an accepting state in the graph, the trivial leading SCCs must be removed. For that, the ELIM-NO-PREDECESSORS procedure is used. Note that after the removal of a leading trivial SCC another trivial SCC may become leading. To deal with that the ELIM-NO-PREDECESSORS procedure proceeds iteratively, and removes all trivial SCCs from the top of the graph (mimicking the topological sort procedure). After ELIM-NO-PREDECESSORS finishes, all leading SCCs in the remaining part of the graph are non-trivial, hence a new round of the elimination is executed, starting again with the ELIM-NO-ACCEPTING procedure.

To learn whether a state is a trivial leading component, the algorithm needs to detect not-yet-removed predecessors of the state. To do so, the algorithm maintains an integer value associated with every vertex to keep the number of not-yet-removed direct predecessors, the so called *indegree*. The unique feature of the OWCTY algorithm is that the indegrees are updated without the need to enumerate the predecessors of a state. In fact, the algorithm only performs forward traversal procedures to maintain the indegrees. This is exactly what the *One-Way* in the name of the algorithm stands for. While this does not immediately make the algorithms on the fly (we do so in the subsequent section), it does already avoid the costly need to store all edges of the graph for reverse traversals as discussed in Section 12.3.1. To emphasize this fact, we again use the implicit definition of the Büchi automaton, i.e., with next-product instead of δ, as defined in Section 12.2.7.

The pseudo code of the Owcty algorithm depends on the following notational conventions. Distributed data structures R, *Open*, and *OldR* are referred to either in a global way, in which case no subscript is used, or in a local (partitioned) way, in which case the subscript denotes which part of a distributed data structure is accessed. For example, a set of states R is a union over distributed data parts of R denoted by $R_1, \ldots, R_n$. R_p is used in procedure Elim-No-Predecessors$_p$ to denote that the algorithm accesses the local part of the data structure. The indegrees are denoted as fields of the states, but in reality should be stored in the state set R_p, which can be implemented as a hash map. At line 20, the indegree is set to 0 for (newly encountered) accepting states, as these are the roots of the search tree, but to 1 for other states, indicating that these are reachable from one accepting state. At line 17, the indegree is incremented when other incoming edges of the state s are found. Termination detection is implemented by Termination.

12.5.2 MAP

Yet another approach to accepting-cycle detection in distributed memory is taken by the algorithm Map [21]. The main idea behind the algorithm is based on the fact that each accepting state lying on an accepting cycle is its own predecessor. When the algorithm computes the set of all accepting predecessors for every accepting state, it is sufficient to check, whether any of the accepting states is present in its own predecessor set. However, to compute and store all this information would be rather expensive. The algorithm instead stores only a single unique representative of the set of all accepting predecessors per state. Let us assume a linear ordering $\prec$ of vertices (given; e.g., by their representation in memory), then the unique representative could just be the *maximal accepting predecessor* (Map). Let $\perp$ be a unique value that is the lowest in the order. We will present here a sequential version of the Map algorithm and explain in a subsequent section how it can be integrated into the Owcty algorithm to achieve a parallel version with on-the-fly properties. See Algorithm 12 for the pseudocode of the sequential version of the Map algorithm.

For a state u, we denote its maximal accepting predecessor in the graph G by $map_G(u)$. Clearly, if an accepting state is its own maximal accepting predecessor ($map_G(u) = u$), then it lies on an accepting cycle. Unfortunately, the converse does not hold in general. Assume that u is the largest accepting state on some accepting cycle. It can happen that the maximal accepting predecessor of u lies outside the cycle, i.e., $map_G(u) = v$ for some accepting state v. However, for this accepting state v either $map_G(v) = v$, in which case the presence of an accepting cycle can be detected on v, or $map_G(v) \prec v$,

Algorithm 11: OWCTY Algorithm

```
global $Open, R, OldR$
function OWCTY($A = (Q, \iota, \text{next-product}, n, M)$)
    $R := Open := \emptyset$
    $o := owner(\iota)$
    $Open_o := \{\iota\}$
    REACH($A, R_p$)
    $OldR := \emptyset$
    while $(R \neq OldR) \wedge (R \neq \emptyset)$ do
        $OldR := R$
        ELIM-NO-ACCEPTING$_1(A, R_1)$ || ... || ELIM-NO-ACCEPTING$_n(A, R_n)$
        ELIM-NO-PREDECESSORS$_1(A, R_1)$ || ... || ELIM-NO-PREDECESSORS$_n(A, R_n)$
    if $R \neq \emptyset$ then report “accepting cycle” else report “no accepting cycle”
function REACH$_p$ ($A = (Q, \iota, \text{next-product}, n, M), R_p$)
    while $Open_p \neq \emptyset \wedge \neg$TERMINATION($Open$) do
        $s := Open_p.dequeue()$
        if $s \in R_p$ then
            $s.indegree := s.indegree + 1$
        else
            $R_p.add(s)$
            $s.indegree :=$ if $M[s] = [1]$ then $0$ else $1$
            forall $t \in \text{next-product}(s)$ do
                $o := owner(t^d)$
                $Open_o.queue(t^d)$
function ELIM-NO-ACCEPTING$_p$ ($A = (Q, \iota, \text{next-product}, n, M), R_p$)
    forall $s \in R_p$ do
        if $M[s] = [1]$ then
            $Open_p.queue(s)$
    $R'_p := \emptyset$
    BARRIER()                    // Wait until all workers reinitialized $R'_p$
    REACH($A, R'_p$)
function ELIM-NO-PREDECESSORS$_p$ ($A = (Q, \iota, \text{next-product}, n, M), R_p$)
    forall $s \in R_p$ do
        if $s.indegree = 0$ then
            $Open_p.queue(s)$
    while $Open_p \neq \emptyset \wedge \neg$TERMINATION($Open$) do
        $s := Open_p.dequeue()$
        $s.indegree := s.indegree - 1$
        if $s.indegree \leq 0$ then
            $R_p.remove(s)$
            forall $t \in \text{next-product}(s)$ do
                $o := owner(t^d)$
                $Open_o.queue(t^d)$
```

Algorithm 12: MAP Algorithm

```
function MAP($A = (Q, \iota, \text{next-product}, n, M)$)
    $Waiting.add(\iota)$
    $oldmap(Q) := \bot$
    $ShrinkM := \emptyset$
    while $Waiting \neq \emptyset$ do
        while $Waiting \neq \emptyset$ do
            $seed := Waiting.dequeue()$
            PROPAGATE-MAP($A, seed, ShrinkM$)
        $Waiting := ShrinkM$
        $ShrinkM := \emptyset$
    report “no accepting cycle”
function PROPAGATE-MAP($A = (Q, \iota, \text{next-product}, n, M), seed, ShrinkM$)
    $oldmap(seed) := seed$
    $map(seed) := \bot$
    $Seeds.queue(seed)$
    while $Seeds \neq \emptyset$ do
        $u := Seeds.dequeue()$
        if $M[u] = [1] \wedge (u \neq oldmap(u))$ then
            if $map(u) \leq u$ then
                $propagate := u$
                $ShrinkM.add(u)$
            else
                $propagate := map(u)$
                $ShrinkM.remove(u)$
        else
            $propagate := map(u)$
        forall $t \in \text{next-product}(u)$ do
            if $propagate = t^d$ then
                report “accepting cycle”
            if $map(t^d) = oldmap(t^s)$ then
                $oldmap(t^d) := oldmap(t^s)$
                $map(t^d) := propagate$
                $Seeds.queue(t^d)$
            else if $(propagate > map(t^d)) \wedge (oldmap(t^d) = oldmap(t^s))$ then
                $map(t^d) := propagate$
                $Seeds.queue(t^d)$
```

in which case v is not part of any cycle in the graph. In the latter case, v can safely be removed from the set of accepting states (or marked as non-accepting). However, removing v from the set of accepting states invalidates the value of $map_G(u)$, which has to be recomputed.

The basic workflow of the algorithm is thus to compute maximal accepting predecessors for accepting states in the graph, and when no accepting cycle can be proved, to *shrink* the set of accepting states. These two steps are alternated until either a cycle is found, or there are no more accepting states in the graph to be removed.

To compute the value of the map_G function, Algorithm 12 proceeds by the principle of value propagation. Note that whenever some value is propagated to a state from which a low value had been propagated before, the new higher value must be repropagated. Due to these duplicate propagations, this procedure requires quadratic time with respect to the size of the graph.

An interesting property of the map_G function is that once computed, the values of map_G partition the graph into subgraphs. More precisely, states that share the same value of map_G may lie on a cycle, however, states that do not share the same value of map_G cannot lie on the same cycle (they cannot be part of the same SCC). The algorithm takes advantage of this observation and in the propagation phase it restricts the propagation to only the subgraphs given by the same value of the map_G function from the previous iteration. In particular, when exploring a transition t within a given subgraph for the first time, it is the case that $map_G(t^d) = oldmap_G(t^s)$ (line 30); later on, $oldmap_G(t^d) = oldmap_G(t^s)$ is used to localise the exploration to a single subgraph (line 34).

To do so, the algorithm maintains *oldmap* values for all states. Also note that the subgraphs where the next iteration of *map* propagation is about to be computed are rooted in the accepting states that were just shrunk. Note that some accepting states may be temporarily recorded as roots of a subgraph, but later on they may become dominated by some other accepting state, in which case they are no longer considered to be roots (see line 24).

An interesting question is how to define the ordering with respect to which the maximal accepting state is determined. It has been shown [22] that for every graph an optimal ordering exists, however, to find it is as difficult as to define a DFS postorder, which is hard to parallelize, and would bring us back to the algorithms in Section 12.4.

12.5.3 Combining OWCTY and MAP

Algorithm MAP works on the fly, i.e., it is capable of reporting the presence of accepting cycles without the need to explore the whole underlying graph. This is not the case with algorithm OWCTY, as to properly compute the indegrees, the whole graph has to be traversed. On the other hand, the time complexity of the OWCTY algorithm is quadratic, while the time complexity of MAP is cubic. In [11] a combination of the two algorithms has been presented to obtain the best of both worlds. In particular, while performing the ELIM-NO-ACCEPTING procedure in the OWCTY algorithm, it is possible to perform limited propagation of *map* values at the same time. The propagation is limited to a single visit of a state (no repropagation is allowed). Still, if the algorithm finds an accepting state that is its own predecessor, the accepting cycle may be reported and the algorithm may terminate without the need for the whole exploration of the graph.

Table 12.1 provides an overview of all emptiness algorithms discussed in this chapter. We use a subjective scale for the scalability of these algorithms, as a theoretical treatise on the matter is out of the scope of this handbook. The DFS algorithms feature optimal runtimes, but not necessarily good scalability. The BFS algorithms on the other hand sacrifice the optimality property to attain better scalability. However, in practice, both approaches have been shown to scale well on multi-core machines [43, 65, 12]. Moreover, in many important cases, i.e., for weak automata, the Owcty algorithms and its combination with Map also achieve optimal runtime in theory.

The integration of Map into the Owcty algorithm further yields some on-the-fly behavior. While not completely on the fly, Owcty tends to deliver shorter counterexamples because of its use of BFS. Short counterexamples are important for repairing errors in the model as they simplify error diagnosis. In practice, Cndfs has been shown to also be able to yield similarly short counterexamples with increasing parallelism [43], but it provides no guarantees about counterexample length. Thus far, only the SCC algorithms are suitable for direct use on TGBAs. Cndfs likely cannot be adapted to support TGBAs without increasing the complexity, but we consider the combination of the BFS algorithms with TGBAs to be an open problem.

12.6 Conclusion

This chapter has revisited the automata-theoretic approach to LTL model checking in Section 12.2. The starting point is a translation of an LTL formula into a (Transition-based Generalized) Büchi Automaton. Fragments of LTL

	Complexity	Scalability	Optimal	On-the-fly	TGBA
Cndfs	$\mathcal{O}(V+E)$	+	Yes	Yes	No
Ufscc	$\mathcal{O}(V+E)$	+	Yes	Yes	Yes
Owcty					
general Büchi	$\mathcal{O}(V.(V+E))$	++	No	No	?
weak Büchi	$\mathcal{O}(V+E)$	++	Yes	No	?
Map	$\mathcal{O}(V^2 \cdot (V+E))$	++	No	Partially	?
Owcty + Map					
general Büchi	$\mathcal{O}(V \cdot (V+E))$	++	No	Partially	?
weak Büchi	$\mathcal{O}(V+E)$	++	Yes	Partially	?

Table 12.1 Overview of distributed-memory algorithms for accepting-cycle detection. Complexity is expressed in the number of vertices V, the number of edges E, and the number of processes P

lead to weak or even terminal automata. The LTL model checking algorithm is reduced to emptiness checking of automata, which boils down to detecting accepting cycles.

To speed up cycle detection, we have introduced parallel algorithms for shared-memory multi-core machines in Section 12.4. These algorithms are based on Depth-First Search and come in two flavors: those based on Nested-DFS, and those based on SCC detection. We showed instances of both.

Based on the observation that DFS is hard to parallelize, an alternative is to design BFS-based algorithms to detect accepting cycles. We have done so in Section 12.5. This type of algorithm is used in shared-memory machines, but was originally designed for distributed clusters of machines connected by a fast communication network.

Although this chapter has focused on the algorithmic ideas behind the various parallel LTL model checking algorithms, we would like to stress that the algorithms that we have explained are also available to the community in open-source tools. The translations from LTL to automata are available in the Spot toolset[3] [36, 38]. Various DFS-based multi-core algorithms are available in the LTSmin toolset[4] [67, 62]. Finally, the distributed and multi-core implementation of the BFS-based algorithms are available through the DiVinE toolset[5] [10, 14].

The scientific papers connected to the algorithms implemented in these tools report on extensive experiments to investigate the practical efficiency and parallel speedup on various benchmark suites of realistic examples, and on their performance in international model checking competitions.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535845. URL http://doi.acm.org/10.1145/2535838.2535845.

[2] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.46.

[3] https://spot.lrde.epita.fr

[4] http://fmt.cs.utwente.nl/tools/ltsmin

[5] https://divine.fi.muni.cz

[3] A. Aggarwal, R. J. Anderson, and M. Kao. Parallel depth-first search in general directed graphs. *SIAM J. Comput.*, 19(2):397–409, 1990. doi: 10.1137/0219025. URL http://dx.doi.org/10.1137/0219025.

[4] R. Alur, S. Chaudhuri, K. Etessami, and P. Madhusudan. On-the-fly reachability and cycle detection for recursive state machines. In N. Halbwachs and L. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 61–76. Springer Berlin Heidelberg, April 2005.

[5] R. Anderson and H. Woll. Wait-free Parallel Algorithms for the Union-find Problem. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 370–380, New York, NY, USA, 1991. ACM. ISBN 0-89791-397-3. doi: 10.1145/103418.103458. URL http://doi.acm.org/10.1145/103418.103458.

[6] T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *Proc. of the 18th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12)*, volume 7214 of *LNCS*, pages 95–109. Springer, 2012.

[7] T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In *Proceedings of the 20th International SPIN Symposium on Model Checking of Software (SPIN'13)*, volume 7976 of *Lecture Notes in Computer Science*, pages 81–98. Springer, July 2013. doi: 10.1007/978-3-642-39176-7˙6.

[8] C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[9] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011. doi: 10.1145/1965724.1965743. URL http://doi.acm.org/10.1145/1965724.1965743.

[10] J. Barnat, L. Brim, and P. Rockai. DiVinE 2.0: High-performance model checking. In *Proceedings of the International Workshop on High Performance Computational Systems Biology (HiBi'09)*, pages 31–32. IEEE Computer Society Press, 2009.

[11] J. Barnat, L. Brim, and P. Ročkai. A time-optimal on-the-fly parallel algorithm for model checking of weak LTL properties. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *LNCS*, pages 407–425, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] J. Barnat, L. Brim, and P. Ročkai. Scalable shared memory LTL model checking. *International Journal on Software Tools for Technology Transfer*, 12(2):139–153, 2010.

[13] J. Barnat, P. Bauch, L. Brim, and M. Ceka. Computing strongly connected components in parallel on cuda. In *2011 IEEE International*

Parallel Distributed Processing Symposium, pages 544–555, May 2011. doi: 10.1109/IPDPS.2011.59.

[14] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser. Divine 3.0 - an explicit-state model checker for multithreaded C & C++ programs. In N. Sharygina and H. Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 863–868. Springer, 2013. ISBN 978-3-642-39798-1. doi: 10.1007/978-3-642-39799-8˙60. URL http://dx.doi.org/10.1007/978-3-642-39799-8˙60.

[15] S. Ben-David, C. Eisner, D. Geist, and Y. Wolfsthal. Model checking at IBM. *Formal Methods in System Design*, 22 (2):101–108, 2003. doi: 10.1023/A:1022905120346. URL http://dx.doi.org/10.1023/A:1022905120346.

[16] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM. ISBN 1-58113-109-7. doi: 10.1145/309847.309942. URL http://doi.acm.org/10.1145/309847.309942.

[17] V. Bloemen and J. van de Pol. Multi-core SCC-Based LTL Model Checking. In R. Bloem and E. Arbel, editors, *Hardware and Software: Verification and Testing: 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, pages 18–33, Cham, 2016. Springer International Publishing. ISBN 978-3-319-49052-6. doi: 10.1007/978-3-319-49052-6˙2. URL http://dx.doi.org/10.1007/978-3-319-49052-6˙2.

[18] V. Bloemen, A. Laarman, and J. van de Pol. Multi-core On-the-fly SCC Decomposition. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 8:1–8:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4092-2. doi: 10.1145/2851141.2851161. URL http://doi.acm.org/10.1145/2851141.2851161.

[19] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[20] D. Bošnački. A nested depth first search algorithm for model checking with symmetry reduction. In D. A. Peled and M. Y. Vardi, editors, *Formal Techniques for Networked and Distributed Sytems*, volume 2529 of *LNCS*, pages 65–80. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-00141-6. doi: 10.1007/3-540-36135-9˙5.

[21] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*,

volume 3312 of *Lecture Notes in Computer Science*, pages 352–366. Springer, November 2004.

[22] L. Brim, I. Černá, P. Moravec, and J. Šimša. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *Proceedings of the 4th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2005)*, pages 1–12, Lisboa, Portugal, 2005. TU Munchen.

[23] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[24] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[25] I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In B. Rovan and P. Vojtáă, editors, *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS'03)*, volume 2747 of *Lecture Notes in Computer Science*, pages 318–327, Bratislava, Slovak Republic, Aug. 2003. Springer-Verlag.

[26] I. Černá and R. Pelánek. Distributed explicit fair cycle detection (set based approach). In T. Ball and S. Rajamani, editors, *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN'03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 49–73. Springer Berlin Heidelberg, May 2003.

[27] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. *Counterexample-Guided Abstraction Refinement*, pages 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45047-4. doi: 10.1007/10722167˙15. URL http://dx.doi.org/10.1007/10722167˙15.

[28] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1): 7–34, 2001.

[29] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. *Symmetry reductions in model checking*, pages 147–158. Springer, 1998. ISBN 978-3-540-69339-0. doi: 10.1007/BFb0028741. URL http://dx.doi.org/10.1007/BFb0028741.

[30] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[31] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithm for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[32] J.-M. Couvreur. On-the-fly verification of temporal logic. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 253–271, Toulouse, France, Sept. 1999. Springer-Verlag. ISBN 3-540-66587-0.

[33] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In P. Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, volume 3639 of *Lecture Notes in Computer Science*, pages 143–158. Springer, Aug. 2005.

[34] A. Deshpande, F. Herbreteau, B. Srivathsan, T. Tran, and I. Walukiewicz. Fast detection of cycles in timed automata. *CoRR*, abs/1410.4509, 2014. URL http://arxiv.org/abs/1410.4509.

[35] E. W. Dijkstra. EWD 376: Finding the maximum strong components in a directed graph. http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD376.PDF, May 1973.

[36] A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13)*, volume 8172 of *Lecture Notes in Computer Science*, pages 442–445, Hanoi, Vietnam, Oct. 2013. Springer. doi: 10.1007/978-3-319-02444-8˙31.

[37] A. Duret-Lutz. LTL translation improvements in Spot 1.0. *International Journal on Critical Computer-Based Systems*, 5(1/2):31–54, Mar. 2014. doi: 10.1504/IJCCBS.2014.059594.

[38] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, 2016.

[39] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, New York, Mar. 1998. ACM Press.

[40] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 8th international Spin workshop on model checking of software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*, pages 57–79. Springer-Verlag, 2001.

[41] E. A. Emerson and T. Wahl. *Dynamic Symmetry Reduction*, pages 382–396. Springer, 2005. ISBN 978-3-540-31980-1. doi: 10.1007/978-3-540-31980-1˙25. URL http://dx.doi.org/10.1007/978-3-540-31980-1˙25.

[42] S. Evangelista, L. Petrucci, and S. Youcef. Parallel nested depth-first searches for LTL model checking. In *Proceedings of the 9th international conference on Automated technology for verification and analysis (ATVA'11)*, volume 6996 of *Lecture Notes in Computer Science*, pages 381–396. Springer-Verlag, 2011.

[43] S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved multi-core nested depth-first search. In *Proceedings of the 10th international conference on Automated technology for verification and analysis*

(ATVA'12), volume 7561 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2012.

[44] K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is there a best symbolic cycle-detection algorithm? In *Proceedings of the fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 420–434. Springer-Verlag, 2001.

[45] L. Fix. Fifteen years of formal property verification in intel. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 139–144. Springer, 2008. ISBN 978-3-540-69849-4. doi: 10.1007/978-3-540-69850-0˙8. URL http://dx.doi.org/10.1007/978-3-540-69850-0˙8.

[46] L. K. Fleischer, B. Hendrickson, and A. Pınar. *On Identifying Strongly Connected Components in Parallel*, pages 505–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-45591-2. doi: 10.1007/3-540-45591-4˙68. URL http://dx.doi.org/10.1007/3-540-45591-4˙68.

[47] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3-4):107–114, February 2000.

[48] A. Gaiser and S. Schwoon. Comparison of algorithms for checking emptiness on Büchi automata. In P. Hlinený, V. Matyás, and T. Vojnar, editors, *Procedings of Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'09)*, volume 13 of *OASICS*. Schloss Dagstuhl, Leibniz-Zentrum fuer Informatik, Germany, Nov. 2009.

[49] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, 2001. Springer-Verlag.

[50] P. Gastin, P. Moro, and M. Zeitoun. Minimization of counterexamples in SPIN. In S. Graf and L. Mounier, editors, *Proceedings of the 11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *Lecture Notes in Computer Science*, pages 92–108, Apr. 2004.

[51] H. Gazit and G. L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Inf. Process. Lett.*, 28(2):61–65, 1988. doi: 10.1016/0020-0190(88)90164-0. URL http://dx.doi.org/10.1016/0020-0190(88)90164-0.

[52] J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, Nov. 2005. Conference paper selected for journal publication.

[53] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulæ to Büchi automata. In D. Peled and M. Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'02)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308–326, Houston, Texas, Nov. 2002. Springer-Verlag.

[54] P. Godefroid. Using partial orders to improve automatic verification methods. In *Computer-Aided Verification*, pages 176–185. Springer, 1991.

[55] P. Godefroid, G. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, Nov. 1995.

[56] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, 1995.

[57] X. He and Y. Yesha. A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs. *SIAM J. Comput.*, 17(3):486–491, 1988. doi: 10.1137/0217028. URL http://dx.doi.org/10.1137/0217028.

[58] G. Holzmann. Parallelizing the spin model checker. In A. Donaldson and D. Parker, editors, *SPIN'12*, volume 7385 of *LNCS*, pages 155–171. Springer, 2012. ISBN 978-3-642-31758-3. URL http://dx.doi.org/10.1007/978-3-642-31759-0_12.

[59] G. J. Holzmann, D. A. Peled, and M. Yannakakis. On nested depth first search. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proceedings of the 2nd Spin Workshop*, volume 32 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science.* American Mathematical Society, May 1996.

[60] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, Nov 2013. doi: 10.1145/2503210.2503246.

[61] T. Junttila. *On the Symmetry Reduction Method for Petri Nets and Similar Formalisms.* PhD thesis, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2003.

[62] G. Kant, A. Laarman, J. Meijer, J. Pol, S. Blom, and T. Dijk. LTSmin: High-performance language-independent model checking. In C. T. Christel Baier, editor, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707. Springer-Berlin, 2015.

[63] S. Katz and D. Peled. *An efficient verification method for parallel and distributed programs*, pages 489–507. Springer, 1989. ISBN 978-3-540-46147-0. doi: 10.1007/BFb0013032. URL http://dx.doi.org/10.1007/BFb0013032.

[64] A. Laarman. *Scalable multi-core model checking*. PhD thesis, University of Twente, 2014.

[65] A. Laarman and J. van de Pol. Variations on multi-core nested depth-first search. In *PDMC'11*, pages 13–28, 2011.

[66] A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-core nested depth-first search. In T. Bultan and P.-A. Hsiung, editors, *Proceedings of the Automated Technology for Verification and Analysis, 9th International Symposium (ATVA'11)*, volume 6996 of *Lecture Notes in Computer Science*, pages 321–335, Taipei, Taiwan, October 2011. Springer.

[67] A. Laarman, J. van de Pol, and M. Weber. Multi-core LTSmin: Marrying modularity and scalability. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NFM 2011, Pasadena, CA, USA*, volume 6617 of *LNCS*, pages 506–511, Berlin, July 2011. Springer. doi: 10.1007/978-3-642-20398-5_40.

[68] A. Laarman, J. van de Pol, and M. Weber. Parallel Recursive State Compression for Free. In A. Groce and M. Musuvathi, editors, *SPIN 2011*, LNCS, pages 38–56, London, July 2011. Springer. URL http://doc.utwente.nl/77024/.

[69] A. W. Laarman and D. Faragó. Improved on-the-fly livelock detection. In G. Brat, N. Rungta, and A. Venet, editors, *NFM 2013*, volume 7871 of *LNCS*, pages 32–47. Springer, 2013. ISBN 978-3-642-38087-7. doi: 10.1007/978-3-642-38088-4˙3.

[70] A. W. Laarman and A. J. Wijs. Partial-Order Reduction for Multi-core LTL Model Checking. In E. Yahav, editor, *HVC 2014*, volume 8855 of *LNCS*, pages 267–283. Springer, 2014. ISBN 978-3-319-13337-9. doi: 10.1007/978-3-319-13338-6˙20. URL http://dx.doi.org/10.1007/978-3-319-13338-6˙20.

[71] A. W. Laarman, J. C. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In N. Sharygina and R. Bloem, editors, *FMCAD 2010*. IEEE Computer Society, 2010. URL http://dl.acm.org/citation.cfm?id=1998496.1998541.

[72] T. Lai and S. Sahni. Anomalies in parallel branch-and-bound algorithms. *Commun. ACM*, 27(6):594–602, 1984. doi: 10.1145/358080.358103. URL http://doi.acm.org/10.1145/358080.358103.

[73] A. Lenharth, D. Nguyen, and K. Pingali. Parallel graph analytics. *Commun. ACM*, 59(5):78–87, Apr. 2016. ISSN 0001-0782. doi: 10.1145/2901919. URL http://doi.acm.org/10.1145/2901919.

[74] G. Lowe. Concurrent depth-first search algorithms based on Tarjan's Algorithm. *International Journal on Software Tools for Technology Transfer*, pages 1–19, 2015. ISSN 1433-2779. doi: 10.1007/s10009-015-0382-1. URL http://dx.doi.org/10.1007/s10009-015-0382-1.

[75] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Let-*

ters, 17(1):5–20, 2007. doi: 10.1142/S0129626407002843. URL http://dx.doi.org/10.1142/S0129626407002843.

[76] Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC'90)*, pages 377–410, New York, NY, USA, 1990. ACM.

[77] K. L. McMillan. *Symbolic Model Checking*, pages 25–60. Springer US, Boston, MA, 1993. ISBN 978-1-4615-3190-6. doi: 10.1007/978-1-4615-3190-6˙3. URL http://dx.doi.org/10.1007/978-1-4615-3190-6˙3.

[78] K. L. McMillan. *Interpolation and SAT-Based Model Checking*, pages 1–13. Springer, Berlin, Heidelberg, 2003. ISBN 978-3-540-45069-6. doi: 10.1007/978-3-540-45069-6˙1. URL http://dx.doi.org/10.1007/978-3-540-45069-6˙1.

[79] I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1(2):56–58, 1971.

[80] P. Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[81] N. V. Rao and V. Kumar. Superlinear speedup in parallel state-space search. *Foundations of Software Technology and Theoretical Computer Science*, pages 161–174, 1988. URL http://dx.doi.org/10.1007/3-540-50517-2˙79.

[82] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[83] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13)*, volume 8312 of *Lecture Notes in Computer Science*, pages 668–682. Springer, Dec. 2013. doi: 10.1007/978-3-642-45221-5˙44.

[84] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit model checking for generalized Büchi automata. *International Journal on Software Tools for Technology Transfer (STTT)*, ??(??):1–21, Apr. 2016.

[85] P. Sanders. Lastverteilungsalgorithmen fur parallele tiefensuche. number 463. In *in Fortschrittsberichte, Reihe 10. VDI.* Verlag, 1997.

[86] W. Schudy. Finding strongly connected components in parallel using o(log2n) reachability queries. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 146–151, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-973-9. doi: 10.1145/1378533.1378560. URL http://doi.acm.org/10.1145/1378533.1378560.

[87] S. Schwoon and J. Esparza. A note on on-the-fly verification algorithms. In N. Halbwachs and L. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and*

Analysis of Systems (TACAS'05), volume 3440 of *Lecture Notes in Computer Science*, Edinburgh, Scotland, Apr. 2005. Springer.

[88] G. M. Slota, S. Rajamanickam, and K. Madduri. Bfs and coloring-based parallel algorithms for strongly connected components and related problems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 550–559, May 2014. doi: 10.1109/IPDPS.2014.64.

[89] U. Stern and D. L. Dill. Combining state space caching and hash compaction. In *In Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, pages 81–90. Shaker Verlag, 1996.

[90] R. Tarjan. Depth-first search and linear graph algorithms. In *Conference records of the 12th Annual IEEE Symposium on Switching and Automata Theory*, pages 114–121. IEEE, Oct. 1971. Later republished [91].

[91] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[92] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, Apr. 1975.

[93] R. E. Tarjan and J. van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, Mar. 1984.

[94] H. Tauriainen. Nested emptiness search for generalized Büchi automata. In *Proceedings of the 4th International Conference on Application of Concurrency to System Design (ACSD'04)*, pages 165–174. IEEE Computer Society, June 2004.

[95] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (ICATPN'91)*, volume 618 of *Lecture Notes in Computer Science*, pages 491–515, London, UK, 1991. Springer-Verlag.

[96] A. Valmari. The state explosion problem. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets 1: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.

[97] M. Y. Vardi. Automata-theoretic model checking revisited. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'07)*, volume 4349 of *Lecture Notes in Computer Science*, Nice, France, Jan. 2007. Springer. Invited paper.

[98] A. Wijs, J.-P. Katoen, and D. Bošnački. *GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components*, pages 310–326. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08867-9. doi: 10.1007/978-3-319-08867-9˙20. URL http://dx.doi.org/10.1007/978-3-319-08867-9˙20.

Index