Parallel Model Checking of ω-Automata

Vincent Bloemen

Formal Methods and Tools, University of Twente
v.bloemen@utwente.nl

Abstract. Specifications for non-terminating reactive systems are described by ω-regular properties. Such properties can be translated in various types of automata, e.g. Büchi, Rabin, and Parity. A model checker can then check for language containment and determine whether the system meets the specification. Checking these automata becomes more complex when introducing probabilities and/or an adversary, e.g. the uncontrollable environment, to the automaton.
Parallel algorithms have become crucial for fully utilizing current hardware systems. With respect to model checking we therefore focus on designing scalable parallel algorithms for emptiness checking.
This research focuses on designing and improving parallel graph searching algorithms for emptiness checking on various types of ω-automata. As a basis, we developed a scalable multi-core on-the-fly algorithm for the detection of strongly connected components (SCCs). Our aim is to contribute to the state-of-the-art techniques in parallel model checking, based on both theoretical complexity analysis and empirical studies on suitable benchmarks.

1 Introduction

Model checking. The automata-theoretic approach to model checking ω-regular properties involves taking the synchronized product of the (negated) property to check and the state space of the system. The resulting product automaton is then checked for language emptiness by searching for an infinite execution that satisfies the *acceptance condition*, which is defined by the ω-automaton. If such an accepting trace is found, the system is able to perform behaviour that is not allowed by the original property, hence we say that a counterexample has been found [6].

Types of ω-automata. An ω-automaton accepts infinite strings which is useful for specifying behaviour in non-terminating systems, i.e. control systems. The acceptance condition can be described in various types of automata, most commonly Büchi, co-Büchi, Rabin, Streett, Parity and Muller (see Section 3 for the definitions). While each type of (nondeterministic) automaton can describe the same property, the sizes of these automata may differ *exponentially*. As a consequence, the choice of automata *could* significantly improve the time to model check. On the other hand, the model checking procedure may also become a lot more complex for such smaller automata.

Chatterjee and Henzinger [5] provide a good overview on different classes of ω-regular properties and how these *1-player* properties can be extended with e.g. adversaries (2-player) and probabilities ($1\frac{1}{2}$ and $2\frac{1}{2}$-player). With an adversary, the automaton is called a *game* and the goal of player 1 is to 'force' the property, i.e. satisfying the property for all possible actions of the adversary.

Parallel model checking. Multi-core architectures have become increasingly more accessible, and the number of CPU cores grows as well. Scalable solutions have been presented to solve the reachability problem [1,11] and the accepting cycle problem [7,16,3,4]. On a 64-core machine, the accepting cycle problem is currently being solved 25 times faster compared to a sequential approach [4]. For many other acceptance conditions, there is limited to non-existing work in parallel solutions.

Motivation. The main motivation of this work is to better understand how model checking can be efficiently applied in a practical sense. Two aspects of importance are how the type of automaton influences the model checking procedure, and how parallelism can be fully exploited in the algorithms. Currently, however, it remains unknown whether a particular type of ω-automaton can be checked efficiently in parallel. While the common approach in practice seems to use Büchi automata for LTL model checking, a Rabin automaton might be a better alternative [17].

Expected contributions. In this research we aim to contribute to scalable parallel solutions for model checking various types of ω-automata. We focus on explicit state on-the-fly graph search algorithms. At present, we designed a scalable multi-core on-the-fly *strongly connected component (SCC)* algorithm [2,3] based on parallel depth-first search (DFS) and concurrent union-find (more on this in Section 2). We consider this algorithm as a basis for the research and have successfully applied it in the context of LTL model checking for Büchi automata [4]. We continue by designing and investigating parallel solutions for other types of automata. This is followed by studying 2-player cases and stochastic instances, e.g. by improving *Maximal End Component (MEC)* decomposition.

2 Strongly Connected Components in Parallel

Preliminaries. For a directed graph $G := \langle V, E \rangle$, two states $v, w \in V$ are *strongly connected* iff there is a path from v to w and also from w to v. A *strongly connected component (SCC)* is defined as the maximal set of states $C \subseteq V$ such that for all states $v, w \in C$, v and w are strongly connected. An SCC is called *trivial* if it consists of a single state v and there is no edge $v \to v \in E$. We further define the notion that C is a *partial* SCC if all states in C are strongly connected, but C is not necessarily maximal.

We assume that the graph is computed *on-the-fly*. This implies that an algorithm initially only has access to the initial state, and can use a function to compute the successor states: $\mathtt{suc}(v) := \{w \in V \mid v \to w \in E\}$.

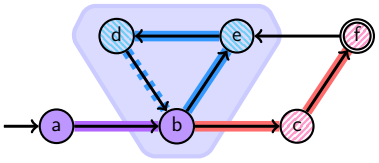

Fig. 1: Two workers cooperate to find an accepting cycle.

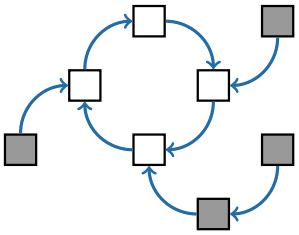

Fig. 2: Cyclic list structure.

A multi-core on-the-fly algorithm for detecting SCCs. The general idea behind the algorithm is to perform multiple *randomized*[1] DFS instances in parallel and globally communicate detected cycles. The main improvement on related work [13,16] is that a complete SCC can be detected in parallel without having to rely on a single worker to visit every state of the SCC. By tracking partial SCCs, multiple workers can even cooperatively detect cycles. As a result, scalable on-the-fly SCC decomposition is now possible for large SCCs.

The algorithm. We describe the algorithm without going in much detail, for a more in-depth description we refer the reader to Bloemen et al. [3]. A concurrent union-find structure is used for globally communicating partial SCCs. In essence this is a structure to maintain sets of states and a single state is the representative or root of a set. Whenever a worker detects a cycle, it merges all (sets of) states on this cycle to a single set in the union-find structure.

Tracking worker IDs. We extended the union-find structure to also maintain worker IDs in the root of each set. When a worker visits a new state v, it adds its worker ID to the root of the set for v. As a consequence, the worker will regard every state in the partial SCC of v as a 'visited' state. We exploit this for detecting cycles, as we show in Fig. 1. Here, a 'blue' worker detected the cycle $\{b, e, d\}$ and the 'red' worker has visited the path $a \to b \to c \to f$. If the red worker visits state e, it will detect that it has already visited this set (namely via b), thus it reports a cycle and merges states c and f to the set.

Cyclic lists for tracking non-fully explored states. We say that a state v is fully explored if all its successors either direct to completed SCCs or to other states in the set of v, since we cannot gain more information from v. A cyclic list, illustrated in Fig. 2, tracks all states in the partial SCC that still have to be fully explored (marked white) and removes the fully explored ones (marked gray). Cyclic lists get merged when states are added to the partial SCC. Workers select states from the list to search from. When the list is empty, all states of the (partial) SCC have been fully explored and the SCC can be marked as completed.

[1] The set of successors is randomly ordered for each worker, such that each worker explores the graph in a different order.

3 Acceptance on ω-automata

An ω-automaton is defined in Definition 1, as presented by Grädel et al. [8]

Definition 1 (ω-automaton). *An ω-automaton is a tuple $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, \mathcal{F} \rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ is the state transition function, $q_0 \in Q$ is the initial state, and $\mathcal{F}$ is the* acceptance component*. In a deterministic ω-automaton, a transition function $\delta : Q \times \Sigma \to Q$ is used.*

Acceptance conditions. We describe the acceptance component $\mathcal{F}$ for different types of ω-automata. A *run* is an infinite sequence of states, starting from q_0, such that for every two successive states v, w there is a transition from v to w. A word $\alpha \in \Sigma^\omega$ is accepted by $\mathcal{A}$ iff there exists a run ρ of $\mathcal{A}$ on α such that

- *(Büchi acceptance)* $\texttt{Inf}(\rho) \cap \mathcal{F} \neq \emptyset$, where $\mathcal{F} \subseteq Q$ is a set of accepting states.
- *(co-Büchi acceptance)* $\texttt{Inf}(\rho) \cap \mathcal{F} = \emptyset$, where $\mathcal{F} \subseteq Q$.
- *(Rabin acceptance)* $\exists (L, R) \in \mathcal{F} : (\texttt{Inf}(\rho) \cap L = \emptyset) \wedge (\texttt{Inf}(\rho) \cap R \neq \emptyset)$, where $\mathcal{F} = \{(L_1, R_1), \ldots, (L_k, R_k)\}$ with $L_i, R_i \subseteq Q$.
- *(Streett acceptance)* $\forall (L, R) \in \mathcal{F} : (\texttt{Inf}(\rho) \cap L \neq \emptyset) \vee (\texttt{Inf}(\rho) \cap R = \emptyset)$, where $\mathcal{F} = \{(L_1, R_1), \ldots, (L_k, R_k)\}$ with $L_i, R_i \subseteq Q$.
- *(Parity acceptance)* $\min\{\mathcal{F}(q) \mid q \in \texttt{Inf}(\rho)\}$ is even, where $\mathcal{F} : Q \to \{1, \ldots, k\}$ is a mapping from states to priorities.
- *(Muller acceptance)* $\texttt{Inf}(\rho) \in \mathcal{F}$, where $\mathcal{F} \subseteq 2^Q$ is a collection of accepting sets of states.

Here, $\texttt{Inf}(\rho)$ denotes the set of states that is visited infinitely often in the run ρ. Fig. 1 illustrates a Büchi automaton, where f is an accepting state and $a \to b \to c \to f \to e \to d \to b \to \ldots$ is an accepting cycle.

Generalized and transition-based acceptance. Conjunctions of multiple Büchi Automata (BA) can also be described with *Generalized Büchi Automata (GBA)*. A GBA considers a set of multiple acceptance conditions, meaning that a run is accepting iff all acceptance conditions are visited infinitely often. Another variant is the *Transition-based Büchi Automata (TBA)* with acceptance on edges instead of states and the combination is called a *Transition-based Generalized Büchi Automata (TGBA)*. Such generalized variants of automata can significantly reduce the state-space, though tracking acceptance becomes more involved. We observed that using a TGBA instead of a BA does not necessarily lead to better model checking performance in practice [4].

4 Related Work

One-player automata. As already mentioned in Section 1, efficient parallel solutions exist for the reachability problem [1,11] and the accepting cycle problem [7,16,3] (or Büchi acceptance). Recently, a GPU algorithm for model checking Rabin automata was presented [17]. Streett acceptance is somewhat related

to fairness detection, a problem for which existing work is present in a parallel setting [12]. For other acceptance conditions no existing work on parallel algorithms seems to exist.

Two-player and stochastic automata. Interestingly, in a 2-player context there is plentiful work on solving Parity acceptance (Parity games) sequentially, but related work also includes a few parallel solutions [15,9]. To the best of our knowledge, no parallel algorithms exist for the remaining automata. Wijs et al. [18] present a solution for MEC decomposition on GPUs, a core problem in stochastic model checking (which also relates to Büchi games).

5 Current and Future Work

Approach. (On-the-fly) SCC detection forms a basis for emptiness checking algorithms. Our plan is to apply our parallel on-the-fly SCC algorithm [3] and related 'building blocks' in detecting the various acceptance conditions. We use these building blocks to parallelize existing techniques, and e.g. in the case of Parity games improve existing work by applying various novel optimizations.

Evaluation. We evaluate the performance and scalability of our algorithms (1) theoretically, using appropriate notions of complexity analysis and (2) empirically, by performing experiments on existing publicly available benchmark suites (e.g. the BEEM [14] database and experiments from the Model Checking Contest [10]) and comparing with related work. We relate this to the original properties to compare different acceptance conditions.

Current stage of research. Currently, one of the four years of the PhD has passed. We have published two papers, one presents the SCC algorithm [3] and another that applies the algorithm for LTL checking with Büchi automata [4]. We also obtained a first place for LTL checking in the 2016 Model Checking Contest [10]. We are currently investigating Rabin and Streett acceptance.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. This work is supported by the 3TU.BSR project.

References

1. Barnat, J., Brim, L., Rockai, P.: DiVinE 2.0: High-Performance Model Checking. In: Proceedings of the 2009 International Workshop on High Performance Computational Systems Biology. pp. 31–32. HIBI '09, IEEE Computer Society (2009)
2. Bloemen, V.: On-The-Fly Parallel Decomposition of Strongly Connected Components. Master's thesis (2015)
3. Bloemen, V., Laarman, A., van de Pol, J.: Multi-core On-the-fly SCC Decomposition. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 8:1–8:12. PPoPP '16, ACM (2016)

4. Bloemen, V., van de Pol, J.: Multi-core SCC-based LTL Model Checking. In: Haifa Verification Conference. Springer (2016), to appear.
5. Chatterjee, K., Henzinger, T.A.: A Survey of Stochastic ω-regular Games. Journal of Computer and System Sciences 78(2), 394–413 (2012)
6. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)
7. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved Multi-Core Nested Depth-First Search. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis, pp. 269–283. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2012)
8. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata Logics, and Infinite Games: A Guide to Current Research. Springer-Verlag (2002)
9. Hoffmann, P., Luttenberger, M.: Solving Parity Games on the GPU. In: Van Hung, D., Ogawa, M. (eds.) Automated Technology for Verification and Analysis, Lecture Notes in Computer Science, vol. 8172, pp. 455–459. Springer International Publishing (2013)
10. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Chiardo, G., Hamez, A., Jezequel, L., Miner, A., Meijer, J., Paviot-Adet, E., Racordon, D., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Trịnh, G., Wolf, K.: Complete Results for the 2016 Edition of the Model Checking Contest (2016)
11. Laarman, A., van de Pol, J., Weber, M.: Boosting Multi-core Reachability Performance with Shared Hash Tables. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design. pp. 247–256. FMCAD '10 (2010)
12. Liu, Y., Sun, J., Dong, J.: Scalable Multi-core Model Checking Fairness Enhanced Systems. In: Breitman, K., Cavalcanti, A. (eds.) Formal Methods and Software Engineering, Lecture Notes in Computer Science, vol. 5885, pp. 426–445. Springer Berlin Heidelberg (2009)
13. Lowe, G.: Concurrent depth-first search algorithms based on Tarjan's Algorithm. International Journal on Software Tools for Technology Transfer pp. 1–19 (2015)
14. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers, pp. 263–267. Springer Berlin Heidelberg (2007)
15. van de Pol, J., Weber, M.: A Multi-Core Solver for Parity Games. Electronic Notes in Theoretical Computer Science 220(2), 19–34 (2008)
16. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Variations on parallel explicit emptiness checks for generalized Büchi automata. International Journal on Software Tools for Technology Transfer pp. 1–21 (2016)
17. Wijs, A.: BFS-Based Model Checking of Linear-Time Properties with an Application on GPUs, pp. 472–493. Springer International Publishing (2016)
18. Wijs, A., Katoen, J.P., Bošnački, D.: GPU-Based Graph Decomposition into Strongly Connected and Maximal End Components. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, Lecture Notes in Computer Science, vol. 8559, pp. 310–326. Springer International Publishing (2014)