Scalable Parallel Model Checking via Monte-Carlo Tree Search

Reed M. Milewicz*
Center for Computing Research
Sandia National Laboratories
1451 Innovation Pkwy SE
Albuquerque, USA 87123
rmilewi@sandia.gov

Simon Poulding*
Software Engineering Research Lab
Blekinge Institute of Technology
Karlskrona, Sweden 37179
simon.poulding@bth.se

DOI: 10.1145/3149485.3149495
http://doi.acm.org/10.1145/3149485.3149495

ABSTRACT

The future of model checking lies in parallel and distributed computing, but parallel graph search algorithms tailored to directed model checking remains an underdeveloped area of research. In this work, we examine the application of parallel Monte Carlo Tree Search algorithms. We demonstrate how exploratory, randomly sampled rollouts of the search space, coordinated through a minimally communicating work-sharing protocol, can enable us to push the boundaries on the scope and scale of problems amenable to serial search.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: [Model Checking]; G.1.6 [**Numerical Analysis**]: [Optimization]; I.6.8 [**Simulation and Modelling**]: [Monte- Carlo]

Keywords

model checking, Monte Carlo methods, parallel search, heuristic search

1. INTRODUCTION

Computing today pervades all aspects of modern life, so much so that it is said that software "is eating the world"[2]. An increasing share of that computation is being carried out by concurrent and parallel algorithms executed on multi-core and distributed systems. However, much work remains to be done in developing practical solutions for the verification of those algorithms, the focus of this work being on directed model checking.

The essential obstacle in all model checking is state space explosion, and directed model checkers address that challenge by the use of heuristic-guided exploration (in conjunction with pruning techniques like partial order reduction) to cut through the myriad of states to find violations. Another related technique in our arsenal is parallel search. Parallel search is both complementary to heuristic search and has a natural duality with metaheuristic optimization (e.g. parallel search under different parameters being functionally similar to parameter tuning); it can push the boundaries on the scale of software that can be verified. However, parallel model checking introduces problems of its own. Graph search in general is non-trivial to parallelize both because it tends to exhibit poor data locality and because of the high cost of synchronization overhead[1]. Finding the right parallelization strategy for a graph search problem requires tailoring the approach to take advantage of the problem structure, and this is what leads us to consider Monte Carlo Tree Search methods for directed model checking.

*Both authors contributed equally to this paper.

Following the first publication on Monte Carlo Tree Search (MCTS) in late 2006, the past decade has yielded an abounding body of research on the subject[10]. MCTS algorithms have led to remarkable successes in a number of domains, most recently enabling the triumph of AlphaGo over world-class Go player Lee Sedol, a major milestone in the history of AI[3]. From a theoretical standpoint, MCTS is appealing because it is a statistical anytime algorithm that rests on a strong foundation of literature on Monte Carlo methods. Moreover, from an implementation standpoint, it is both memory efficient and readily parallelizable, well-suited for multi-core and distributed applications[8][5].

In this work, we present a MCTS-based parallel search strategy for directed model checking using Java Pathfinder. We build upon the findings of Poulding and Feldt 2015[22], which first explored the application of heuristic-guided MCTS in this domain. While previous research yielded promising results, it only considered the serial search case, and realizing MCTS for model checking in the parallel case is challenging in its own right. The contributions of this work are as follows: (1) a minimally communicating parallel search strategy that divides labor between one or more master instances which drive the search and look-ahead simulation instances which randomly sample yet unexplored regions of the state space, and (2) experimental results which demonstrate the effectiveness of our approach.

The rest of the paper is laid out as follows. In section 2, we present background information and prior work on parallel model checking and MCTS. In section 3, we give a detailed description of the implementation of our search algorithm, and in section 4 presents experimental results. Finally, in section 5, we lay out avenues for future research.

This paper is dedicated to the memory of joint co-author Simon Poulding, who passed away unexpectedly days before its completion.

2. BACKGROUND

The future of model checking — indeed, all formal verification — lies in parallel and distributed computing. As Camilli 2015 has observed, the technological shift towards these forms of computing has guaranteed that the next great leaps forward in efficiency for verification will depend upon them[6]. This fact was never lost on the model checking community, but realizing that vision is easier said than done. Research into parallel model checking began in earnest in the early-to-mid 2000s, and was seen as promising, but it quickly became apparent that there would be persistent, fundamental challenges, namely communication overhead[26][16].

Ideally, we would like to limit redundant computation of states so as to maximize the amount of useful parallel work. The naïve

solution is to keep a globally synchronized set of "seen" states, but this often ends up being prohibitively expensive. In response to this, a number of papers followed that attempted to realize non-communicating or minimally communicating parallel search in model checking[12][17][24][13][23][4]. These include strategies such as dividing up the search into non-overlapping partitions or executing many independent searches in parallel. Non-communicating parallelism, however, has limits of its own, such as the fact that, for sufficiently large and irreducible problems, the memory limits of an individual machine becomes a barrier without better heuristics or further limiting search completeness.

These conditions, we argue, present a fertile ground for the development of new model checking search techniques based on MCTS-style approaches. The value of random sampling, which lies at the core of Monte Carlo methods, has been established in the literature. We call particular attention to Dwyer et al. 2006[12], an exploratory study of randomized parallel search applied to model checking, which found that random search could deliver competitive results given adequate computing power, and Grosu and Smolka 2005, which demonstrated that Monte Carlo techniques could be used to set bounds on the likelihood of randomized walks uncovering an error[15]. However, as noted by Pelánek et al. 2005, pure random search can easily become trapped or fail to sufficiently explore common classes of state spaces such those with many diamond-like structures (e.g. mutual exclusion algorithms)[21]. The authors of that work argue that random search can be enhanced by using, among other things, periodic re-initialization of random walks, sampling multiple walks, and augmenting the search with heuristic guidance; MCTS allows for all of these features.

MCTS algorithms use random sampling to carry out simulated explorations or rollouts of the search space ahead of the current position of the search. As demonstrated by Chaslot et al. 2008, there are numerous ways in which rollouts can be parallelized[8], the intermediate results of which can either be cached or eliminated, depending on resource constraints. Likewise, it is possible to prune subtrees which did not yield promising rollouts, giving yet another way to manage the resource consumption. Moreover, MCTS search algorithms are easily paired with all manner of heuristics to shape the search.

3. IMPLEMENTATION

3.1 Nested Monte-Carlo Search

Arguably the most successful application of Monte-Carlo tree search algorithms is playing one side of complex two-player games such as Go. We may equate the search for property violations by a concurrent program to a game in which a program state is equivalent to a game state, and the transitions out of a program state are the 'moves' that the player may make. However, the analogy is to a single-player rather than two-player game since during the search for a property violation, there is no equivalent of an opponent who makes competing moves towards an objective that is the opposite of our own.

In this work, we apply a form of MCTS called Nested Monte-Carlo Search (NMCS) originally proposed by Cazenave and demonstrated to be effective on single-player games[7]. The lookahead policy is defined recursively given a nesting parameter $N \in \mathbb{Z}_0^+$ and is straight-forward. At the current state, exactly one simulation is performed for each possible move. If the nesting factor is zero, then this means making random moves until an end state is reached. Otherwise, the simulation carries out its own instance of NMCS with a nesting factor of $N-1$ on that move's decendants. This use of nesting is intended to reach a balance between exploration of the graph and exploitation of promising subtrees without the need for parameter tuning.

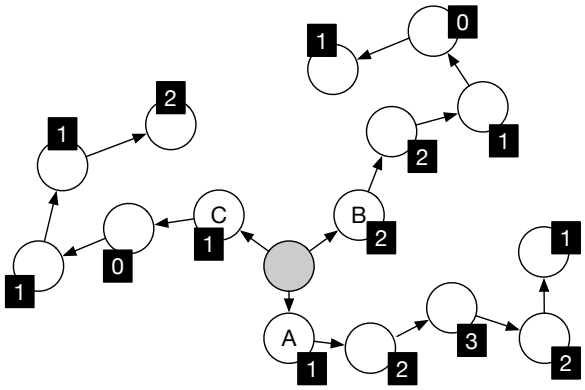

Figure 1: **An illustration of one iteration of the Nested Monte-Carlo Search algorithm. The central grey circle represents the current state, from which there are three possible transitions to states A, B, and C. The paths from these three states are simulations. The reward heuristic for each state on the simulation path is shown in the black square.**

The complexity of conventional NMCS is $O(b^N h^{N+1})$ where b is the branching factor and h is the height of the tree[18]. In Poulding and Feldt 2015, one of the authors of this paper proposed an enhancement to the algorithm (which we will call h_l) that limits the number of moves taken by each simulation [22]. This was for the sake of efficiency: reaching the end state when model-checking may require very many random moves. Results revealed that looking ahead a smaller number of states could still provide adequate guidance to the search. In this work, we have also noted that the branching factor, which for our problems of interest can excceed what is common for many games, is a potential bottleneck in a parallel implementation as each worker can end up with a backlog of subtrees to expand. For this reason, we also offer a parameter (b_l) that permits the search to only expand a limited, randomly selected number of descendants at each depth.

Additionally, in the application of the algorithm described here, we make a further enhancement. Instead of reporting the value of the last state visited during the simulation, as is done with standard NMCS, the heuristic is calculated at each state visited during the simulation, and the reward used to choose the move taken by the algorithm is the best value over the path of the simulation. The motivation for this enhancement is that if a simulation traverses through a region of the state space where states have a high value of heuristic but ends in a lower value region, then the simulation is rewarded on the basis that the move *could* lead to the high-value region.

Figure 1 illustrates one iteration of algorithm. From the current state (the circle filled in grey), there are three possible transitions leading to states A, B, and C. During this iteration, the algorithm must decide which of these three states becomes the current state at the next iteration. To do this, one simulation is performed from each of the three states following a path which, in this case, is parameterized to be at most 4 transitions long; for clarity, possible states other than those actually visited by the simulations are not shown in the figure. At each state on the simulation path,

the reward heuristic is measured, and this is shown in the black square annotating the states. In this example, higher rewards are better. The simulation starting from state A encounters the best reward value (3) during the simulations, and so the transition to A is taken and A becomes the current state in the next iteration of the algorithm. Note that that a best-first search (equivalent to NMCS with a simulation path length of 0) would have chosen to move to state B since it has the best reward value of the states A, B, C; while if the reward were measured only at the state at end of the simulation during NMCS, then state C would have been chosen.

3.2 Parallelization

The independent simulations performed by Monte Carlo tree search algorithm present an opportunity for parallelization. This is particularly true for NMCS in which at each iteration the number of simulations and their starting states are fixed in advance; in other Monte-Carlo tree search algorithms the result of one simulation determines the starting state for the subsequent simulations. In this paper, we investigate such a parallelized configuration in which a master JPF search process distributes the work of the simulations to separate slave JPF processes. When the number of transitions from each state is large and/or the simulation path length is long, such a configuration may permit the effective utilization of parallel computing architectures to search large state spaces.

3.3 JPF Implementation

We implemented Nested Monte-Carlo Search in JPF as a subclass of `gov.nasa.jpf.search.Search`. We note that NMCS requires very few model states to be stored and so has a relatively low memory requirement: in our implementation we explicitly save only the 'current' state so that JPF may quickly return to this state after performing a simulation.

When the search is parallelized, the master and multiple slave search processes are separate JPF instances running in their own JVM (for example, on different host machines). In order to exchange state information between the master and slaves, one option would have been to serialize and then send the current VM state. However, for any reasonably large program-under-test, this state would be large and therefore incur a significant timing overhead that could limit the efficiency of the parallelized search.

Instead, we identify states by the sequence of moves taken to reach it from the initial root state , i.e. the indices of transitions chosen from each choice generator. The randomization of choice generators is disabled so that the ordering of the choices for each choice generator is the same for both master and slave, and so only the sequence of choice indices is necessary to identify the same state in each of the JPF instances.

For large models, this sequence could be long and therefore might still incur a significant timing overhead in not only communicating the sequence but also moving the slave to the correct state in order to run a simulation by applying the sequence of transitions. However, if we ensure that a slave's current state is always a state on the path from the root state to the master's current state (i.e. either the master's current state or a current state it had at an earlier iteration of the search), it is easy to identify and communicate the (in general) short subpath between the slave's current state and the master's current state. The slave can then easily move to the same current state as the master by following the (in general) small number choices in this subpath. After running a simulation, the slave restores its state to the last-known master state so as to ensure the slave's current state is on the path to the master's current state.

The communication between master and slave JPF processes is implemented using Java remote method invocation (RMI) technology. In this configuration, the master search process is an RMI server that listens for connections from slave search processes that are RMI clients. The connections from slave processes are to request new simulations to run and to return the results of these simulations.

4. EVALUATION

In this section, we present preliminary experimental results that showcase the application of our approach. We begin with our benchmark of choice, the canonical Dining Philosophers problem, using the implementation provided by the SIR repository[11]. We chose this benchmark because the state space is non-trivial for an uninformed search (e.g. BFS, purely random) to navigate due to the (1) high branching factor, (2) the fact there is only one deadlock state that requires at least N steps to reach, and (3) the search can move away from a deadlock state by allowing a philosopher to acquire both locks that it needs. Realistically, the mutual exclusion protocol described by this benchmark is more complex than what is encountered in the wild, but it allows us to simulate searches of large, heavy-weight programs with latent bugs that are unlikely to be found by pure chance.

Our baseline heuristic (and the guiding heuristic used by the NMCS algorithm) is the most-blocked structural heuristic which comes standard with JPF and was first described in Groce and Visser 2002[14]:

$$h_{mostblocked} = N_{alive} - N_{runnable} \quad (1)$$

This is an intuitive choice: maximizing the number of blocked threads leads us to a state where all threads are unable to make progress. The most-blocked heuristic performs favorably on this benchmark, with the time/space requirements to find the bug being linear with respect to the number of philosophers. This can only carry us so far: as we pile on many hundreds of philosophers, the memory requirements for storing all the states begins to exceed the limits of RAM on commodity hardware. Fortunately, as we can see on Figure 2, there remains a great deal of room for improvement.

Borrowing a page from Milewicz 2016[20], we throw the kitchen sink at the problem: we apply an estimation-of-distribution of algorithm by sampling small versions of it and then use linear regression to craft a weighted combination of various heuristics (including most-blocked)[25], upon which we apply an ant colony optimization meta-heuristic[9], and upon which we then apply a meta-evolutionary hyperparameter tuner on a handful of trial runs. The resulting heuristic brings us much closer to optimal performance.

Alternatively, we can apply NMCS (one master, four workers, nesting limit of one, simulation depth of four) to the most-blocked heuristic, and we can achieve competitive results without the need for prior information or iterative fine-tuning (though our approach could incorporate these as well). However, showcasing those results without qualifying them would be extremely disingenuous. Our implementation requires substantially more computation overall, introduces communication overhead, and may require restarts that further degrade time performance. With

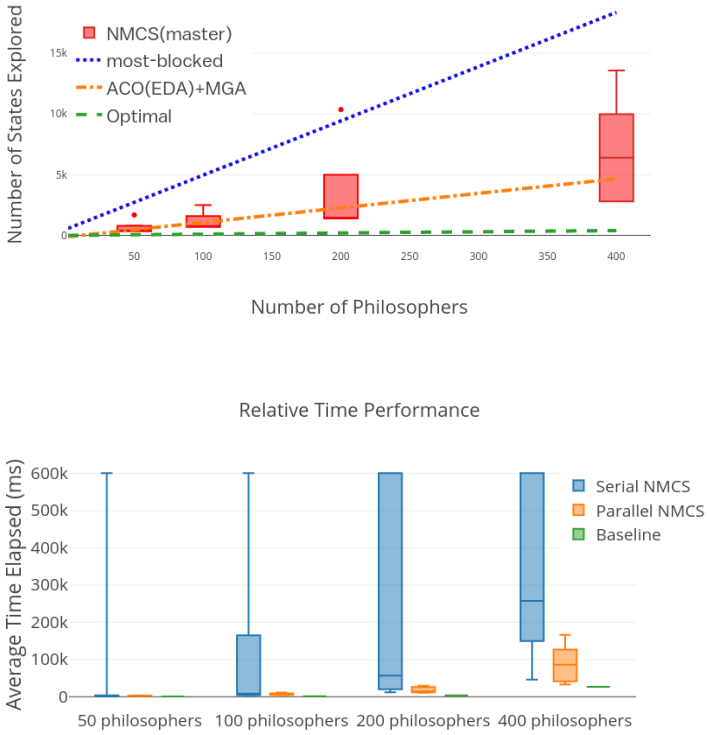

Figure 2: **Above, a graph illustrating the range of performance of NMCS from the perspective of the master instance with comparisons to serial explorations using the baseline heuristic and a hyper-meta-heuristic. The results show that augmenting heuristic search with Monte Carlo sampling can be very powerful. Below, a graph of the time elapsed by runs of parallel and serial variants of our NMCS implementation.**

this in mind, we will detail the results of experiments designed to test the boundaries on which NMCS can accomplish.

We performed our experiments on a cluster of eight 20-core Intel Xeon E5-2698 v4 nodes each with 32 GB of available RAM, made available by the College of Saint Benedict & Saint John's University. We used a setup of one master and four workers, each located on different machines. We ran JPF on the Dining Philosophers problem with thread counts of 50, 100, 200, and 400, we tested with simulation depths and choice limits of 4, 8, and 16, and a heap limit of 6 GB per search instance. Because the outcome of the NMCS is partially random, we repeated each run 10 times with a timeout limit of 600 seconds. Additionally, we compared results between serial and parallel executions of NMCS. We describe the results below.

As expected, the parallel implementation of NMCS is considerably faster than the serial version. Across all experiments, the parallel implementation ran 16.73x faster on average. A breakdown by problem size (from 50 to 400 threads) has the parallel implementation 22.84x, 22.53x, 16.55x, and 5x faster than the serial; this is to say that for a fixed number of workers against an increasing workload, the speed-up is expected to be roughly proportional to the number of workers, the simulation step being embarassingly parallel. Moreover, 99% of all parallel runs completed within the (generous) time limit, compared to 65% of serial runs.

The performance of our NMCS variant is modulated by the choice limit (b_l), which constraints the number of simulations that the master can request at a search step, and the simulation depth (h_l), which constrains the amount of computation involved in a simulation. Here we observed two trends. First, in the absence of b_l (no constraint with respect to branching factor), the parallel speed-up evaporates completely as the philosopher thread count increases; this is because, for a fixed number of workers, increasing the branching factor means more convoying of simulation requests and a growing backlog of simulations to be completed. However, there was no correlation between the choice of b_l and the relative performance of the search; failing to explore subtrees is counterbalanced by the fact that the search can repeat itself. As for the choice of simulation depth, it is worth stressing that more is not always better. Clearly, a simulation depth of 1 has the master requesting workers to evaluate a single state and report the heuristic value, which is not efficient. However, we found that for the particular benchmark in question, the best results were seen at a simulation depth of 4. Aside from the fact that deeper simulations mean that the master spends more time waiting on workers, we only return the best heuristic value of states visited in a rollout, and this means that larger simulation depths are not guaranteed to be more useful (e.g. at a simulation depth of 64, an optimally interesting state 63 states away and another 5 states away are equally appealing). In any case, that there is an optimal value for the simulation depth hints at the possibility of using some form of online parameter tuning to converge on that value.

With regards to the average number of states explored by the master instance under NMCS is 4x lower than that of the baseline heuristic search. However, the number of states explored by the master is only (on average) 5% of the total number of states explored by the master and its workers combined. Overall, our implementation of NMCS explores 5x more states than the baseline search, but the majority of that computation is delegated to workers performing simulations, the intermediate results of which do not need to be retained by the master.

Finally, in terms of counterexample length, a measure of solution quality, the results for NMCS for all choices of parameters and all runs are virtually identical to those of the plain, most-blocked heuristic. This is to say that while look-ahead sampling reduces the amount of unnecessary exploration by the master instance, the search is still driven by the heuristic and is likely to produce solutions of similar quality.

5. CONCLUSION

In this paper, we have presented a translation of parallel Monte Carlo Tree Search to the domain of directed model checking. Our preliminary results are promising, but we have merely scratched the surface of what is possible. It is likely the case that communication between instances could be streamlined so as to reduce overhead further; we can envision schemes in which simulations by workers and exploration by the master are overlapped, maximizing the amount of useful parallel work.

It is also clear that our approach to MCTS could be further improved with the use of meta-heuristics and parameter tuning strategies. In particular, our future work aims to augment random sampling with heuristics derived from prior static analysis ala Milewicz and Pirkelbauer 2017[19].

6. REFERENCES

[1] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David 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*, pages 1–11. IEEE Computer Society, 2010.

[2] Marc Andreessen. Why software is eating the world. *The Wall Street Journal*, 20(2011):C2, 2011.

[3] S Borowiec and T Lien. Alphago beats human go champ in milestone for artificial intelligence. *Los Angeles Times*, 12, 2016.

[4] Mohand Cherif Boukala and Laure Petrucci. Distributed model-checking and counterexample search for ctl logic. *International Journal of Critical Computer-Based Systems 3*, 3(1-2):44–59, 2012.

[5] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[6] Matteo Camilli. Coping with the state explosion problem in formal methods: Advanced abstraction techniques and big data approaches. 2015.

[7] Tristan Cazenave. Nested Monte-Carlo search. In *Proc. 21st Int'l Joint Conf. Artificial Intelligence (IJCAI)*, pages 456–461, 2009.

[8] Guillaume Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. *Computers and Games*, 5131:60–71, 2008.

[9] Francisco Chicano and Enrique Alba. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. *Information Processing Letters*, 106(6):221–231, 2008.

[10] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International Conference on Computers and Games*, pages 72–83. Springer, 2006.

[11] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[12] Matthew B Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *Proceedings of the 29th international conference on Software Engineering*, pages 3–12. IEEE Computer Society, 2007.

[13] Diego Funes, Junaid Haroon Siddiqui, and Sarfraz Khurshid. Ranged model checking. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[14] Alex Groce and Willem Visser. Model checking java programs using structural heuristics. In *ACM SIGSOFT software engineering notes*, volume 27, pages 12–21. ACM, 2002.

[15] Radu Grosu and Scott A Smolka. Monte carlo model checking. In *TACAS*, volume 3440, pages 271–286. Springer, 2005.

[16] Gerard J Holzmann and Dragan Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10), 2007.

[17] Gerard J Holzmann, Rajeev Joshi, and Alex Groce. Tackling large verification problems with the swarm tool. In *International SPIN Workshop on Model Checking of Software*, pages 134–143. Springer, 2008.

[18] Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

[19] Reed Milewicz and Peter Pirkelbauer. Ariadne: Hybridizing directed model checking and static analysis. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 442–447. IEEE, 2017.

[20] Reed Morgan Milewicz. *Improving the scalability of directed model checking of concurrent java code through hybrid and distributed analysis*. PhD thesis, The University of Alabama at Birmingham, 2016.

[21] Radek Pelánek, Tomáš Hanžl, Ivana Černá, and Luboš Brim. Enhancing random walk state space exploration. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, pages 98–105. ACM, 2005.

[22] Simon Poulding and Robert Feldt. Heuristic model checking using a monte-carlo tree search algorithm. In *Proc. 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1359–1366, 2015.

[23] Junaid Haroon Siddiqui and Sarfraz Khurshid. Scaling symbolic execution using staged analysis. *Innovations in Systems and Software Engineering*, 9(2):119–131, 2013.

[24] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194. ACM, 2010.

[25] Jan Staunton and John A Clark. Searching for safety violations using estimation of distribution algorithms. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 212–221. IEEE, 2010.

[26] Ulrich Stern and David L Dill. Parallelizing the murϕ verifier. *Formal Methods in System Design*, 18(2):117–129, 2001.