Effectively using Search-Based Software Engineering Techniques within Model Checking and Its Applications

Jeremy S. Bradbury, David Kelk and Mark Green
Software Quality Research Laboratory
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
{jeremy.bradbury, david.kelk, mark.green}@uoit.ca

***Abstract*—In this position paper, we affirm that there are synergies to be gained by using search-based techniques within software model checking. We will show from the literature how meta-heuristic search based techniques can augment both the model checking process and its applications. We will provide evidence to support this assertion in the form of existing research work and open problems that may benefit from combining Search-Based Software Engineering (SBSE) techniques and software model checking.**

***Index Terms*—model checking, state space, search based software engineering.**

I. Introduction

Software model checking is a verification and debugging technique. It relies on building a finite-state model of the software that is verified against a specification by searching the state space. Software model checking is particularly effective when used with concurrent and distributed programs. It can detect data races, deadlocks and other problems inherent in parallelization. Although model checking is beneficial when used with concurrency it does have limitations in its ability to scale as the amount of parallelization in a given program increases. Specifically, as the number of threads in a concurrent program increase the state space of the corresponding model grows exponentially. This is commonly referred to as the *state explosion problem* [1].

In Search Based Software Engineering (SBSE), meta-heuristic search techniques like genetic algorithms and evolutionary strategies are used to solve a wide array of software challenges. These challenges include (but are not limited to): project planning, maintenance, reverse engineering, source code comprehension, source code refactoring and program repair. Currently the largest application area of SBSE is software testing [2] where SBSE techniques exist that can generate, improve and optimize test suites.

Based on our experience with both model checking and SBSE we feel strongly that model checking will reap dividends when augmented with search-based techniques. In this position paper we will demonstrate the synergies of software model checking and SBSE by exploring the open research opportunities in two distinct but related contexts:

- How SBSE can be used to improve the model checking process
- How SBSE and model checking can be used together to address common Software Engineering problems in an automatic way.

In the next section we briefly introduce and describe software model checking and SBSE. In Section III we will explain the application of SBSE techniques to the model checking process. We briefly survey which research problems have already been address in the literature and which problems remain open. In Section IV we consider common Software Engineering research problems that can benefit from the combined use of SBSE and model checking. We conclude in Section V with a summary of our position and a discussion of open problems and future work involving the combined use of model checking and SBSE.

II. Background

A. Search-Based Software Engineering

Many software engineering challenges can be expressed as optimization problems (e.g., optimizing cohesiveness or optimizing the size of a test suite size). SBSE positions these problems in a search-based context (e.g., search for the optimal cohesiveness or search for the optimal test suite prioritization). In general, a problem representable as a member of an evolvable population (e.g., program correctness) and expressible in terms of a score (e.g., a fitness function to measure the correctness of the program) may be adaptable to search based techniques.

A common feature of SBSE problems is their complexity. In many cases, the use of deterministic algorithms can either take too long to solve or do not exist. IN SBSE, approximate search algorithms are routinely used to make the search tractable. Hill climbing [3] is one example as it uses a greedy strategy of examining all nearest neighbours in the search space and accepting the best one as the new solution. This strategy continues until none of the nearest neighbours improve upon the solution.

Another SBSE technique, particle swarm optimization, is an example of a heuristic, or guided random search patterned on

the flocking behaviour of animals [4]. Every particle encodes a solution to the problem. The movement of members of the swarm is guided by the leader (i.e., the most fit current solution) and the random velocities of each particle in the search space.

Genetic algorithms is another SBSE technique modelled after biological evolution [5]. Evolution is simulated by randomly mutating parts of members, preferentially choosing fitter (higher scoring) solutions for reproduction and preferentially choosing fitter members for inclusion in the next generation.

The element of randomness in heuristic search eliminates determinism and repeatability. From one invocation of a search to another, one doesn't receive the same quality of answer. SBSE is durable in this case. Near optimal solutions or optimizations are better than nothing at all.

B. Model Checking

"Software model checking is the algorithmic analysis of programs to prove properties of their executions. It traces its roots to logic and theorem proving, both to provide the conceptual framework in which to formalize the fundamental questions and to provide algorithmic procedures for the analysis of logical questions" [6]. In other words, software model checking can formally prove a program has certain properties like deadlock freedom and that division by zero never occurs. software model checking simulates the execution of a program by exhaustive enumeration of all of the states and transitions of the program. In each state, all of the values of variables are checked against the software specification (e.g., safety and liveness properties) also any generally desired properties are also verified (e.g., deadlock and data race freedom).

Model checking a software program has several possible outcomes:

1) The software model satisfies the software specification.
2) The software model does not satisfy the specification. In this case, a counter-example is usually provided. For example, if the software is not free of deadlocks then the counter example will be an execution path leading to a deadlock.
3) The model checker runs out of resources. This result can be attributed to the *state space explosion problem* where the number and size of states exceeds the memory of the computer.

Initially software model checking was most often used in the design phase of a software project. High-level designs are more abstract than programs and have smaller state spaces. As model checking matured and computer hardware advanced, software model checkers like Java Pathfinder (JPF) [7] and Bogor [8] were developed that enabled practitioners to directly model check source code artifacts. The reasons for model checking programs include [7]:

- Errors can exist in programs regardless of model checking the designs.
- Critical section errors and deadlocks are introduced at a deeper level of detail than in the design document.
- The need to support debugging and error location using model checking.

Modern model checkers contain a number of features to combat the state space explosion problem including partial order reduction [9], state space collapse [7], slicing and data abstraction. When techniques like these are insufficient for taming the search space, the certainty of model checking can be sacrificed. Only exhaustive search can prove a property true in a software model checker. However, approximate techniques like a heuristic search of the search space [9], [10] can give us confidence a property holds, or a counter-example if it does not.

III. Using SBSE to improve the Software Model Checking Process

In this section we consider the opportunities to leverage the benefits of SBSE's meta-heuristic techniques to enhance the software model checking process. Specifically, we consider how SBSE techniques can enhance model checkers like Java PathFinder (JPF), an open source tool developed by NASA. JPF and similar software model checkers play an important role as debugging tools for concurrent programs. Due to the state explosion problem it is often difficult to prove that large-scale concurrent software is deadlock free or data race free. However, it is possible with JPF to identify execution paths in a program that lead to a deadlock or data race. One of the ways that JPF addresses the inability to perform an exhaustive search of the entire state space of a program is by using heuristic search algorithms that can quickly find "buggy" paths in a program. For example, a heuristic search for detecting hard to find "heisenbugs" was implemented in JPF [11]. There are two topics that we feel have potential for future research in this area: optimizing the state space search of a single model checking run and optimizing the use of incremental model checking as a form of regression testing.

A. Optimizing the State-Space Search of a Single Software Model Checking Run

We believe that exploring a state space in model checking has synergy with the meta-heuristic search techniques utilized in SBSE. This is confirmed by the work of Staunton and Clark who extended the JPF model checker to include a bug detection approach based on the SBSE technique Estimation of Distribution Algorithms (EDAs) [12]. Their experimental results showed that EDA was particularly effective on larger programs. The Staunton and Clark work demonstrates that SBSE is a feasible approach to search in model checking. There are still other opportunities to combine SBSE and software model checking in the context. For example:

- Leverage SBSE to optimize the model checking search algorithm to the program under analysis or to a set of domain-specific programs.
- Leverage SBSE to optimize the model checking search algorithm to specific bug patterns.

B. Optimizing the Use of Incremental Model Checking

Incremental model checking [13], [14], is a technique for more efficiently verifying a program when small changes are made to it over successive iterations. For example, consider the incremental model checking approach described by Lauterburg, et al. [14] . In the first iteration a program is model checked in the standard way and information on the state space and transitions is recorded to a file. Changes are made to the program code and the program is model checked again. The output of the *first iteration* and the modified source code are the inputs to the second model checking iteration. During the second iteration, states are searched for in the data from the first iteration and then in the list of visited states encountered during the second run. Three scenarios may result during the second run's state exploration:

1) If the state is in the new visited states list, this state has already been visited in this run and the model checker doesn't have to evaluate this state again or visit any of it's child states.
2) If the state is found in the previous iteration's list then the model checker also does not have to re-evaluate this state again, but it cannot assume that all child states reached from this state will be the same as before. For example, a change "further down" may result in different child nodes. In this case, effort is still saved by not re-evaluating this state. This is the innovation of incremental modelling and the source of the performance boost. If changes between program versions are small this case should be very common.
3) If the state isn't in either the previous iteration's output or the new visited states list then it is a new state added since the previous run. Thus, it must be evaluated (and added to the new visited list) and all its child states must be explored. If the changes between program versions are small, this case should happen rarely.

The increased use of software model checking as a debugging and testing tool increases the likelihood of it being used iteratively. Therefore, an interesting opportunity for future research is if further optimizations can be achieved by SBSE techniques during incremental model checking. For example, to develop a heuristic search algorithm that visits all of the new states (see scenario 3 above) sooner.

IV. Using SBSE and Software Model Checking to Solve Software Engineering Problems

One way software model checking can be integrated with search based techniques is to use the model checker to determine fitness. Any property that can be model checked (e.g., reachability) can be turned into a measure of fitness (e.g., number of reachable states). Alternatively, model checking can also produce data to be consumed by a SBSE algorithm. One example of this producer-consumer relationship is test generation by model checking that are consumed by software engineering applications such as test prioritization or test suite optimization. Avida-MDE [15] used digital evolution to generate models for autonomic behaviour. A model checker is used to evaluate how well the UML model adheres to formal properties. A previously developed tool [16] that used genetic programming (GP) with model checking to determine fitness is improved to evolve more realistic mutual exclusion algorithms. GP and model checking are combined to both find and fix errors in nontrivial multi-party communications protocols [17].

A. Optimization of Automatically Generated Test Suites

There is a rich body of work around generating test suites with model checkers and separately with SBSE techniques.

For example, using model checking and symbolic execution to generate tests for structural coverage of code with complex data structures [18]. A reachability algorithm was developed in for path coverage that adds tests only if they are not redundant at the global level [19]. Optimization of test suites generated by model checking [20]. A different kind of optimization of test generation where linear temporal logic rewriting was used within a software model checker to improve performance and reduce test suite size [21]. Although the topic of test suite generation has been well researched, there may be further opportunities to leverage model checking and SBSE together to gain new insights into this research problem.

B. Fixing Bugs in Parallel Programs

Great strides have been made in fixing bugs in single-threaded programs [22] [23]. In them, genetic programming was used to evolve patches, while testing evaluates fitness. These techniques cannot be applied directly to fix bugs (i.e., data races and deadlocks) in concurrent programs because of the random nature of thread scheduling and because in any given execution the interleavings leading to the bugs may not be executed.

We believe that fixing data races and deadlocks in parallel programs will be of increasing importance in the multi-core era. Similar to the combined use of SBSE and testing in the above techniques, we believe SBSE and software model checking can be used together to repair concurrency bugs. In other words, model checking could be used instead of testing to determine the fitness of a proposed fix for concurrency bugs. As the full state space is explored the model checker can tell us that a data race or deadlock exists or not.

Due to state space explosion, finding a counter-example that leads to a deadlock or data race may of course be non-trivial. We believe a heuristic search of the state space will guide the model checker to any data races or deadlocks by exploring fewer states than a deterministic search like depthth-first. Even if fewer states are explored, the evaluation of each state is still resource intensive. Here we propose using incremental model checking techniques to lessen this burden. Deadlocks and data races involve modifying synchronization. This doesn't change the structure of the state space from generation to generation, so most states should have been seen before in a previous run and thus not need to be re-evaluated.

Another optimization is to record counter-examples and try them again on every proposed fix. As heuristic search

techniques are not guaranteed to improve a proposed fix from generation to generation, this avoids invoking most of the heavy machinery of model checking.

When no counter-example is generated the heuristic search may have found fixes for the data races and deadlocks. Now it is necessary to perform a full state space search to verify this. The burden is lessened by the use of incremental modelling however, this search may still be impractical in terms of time or resources or fail because of the state space explosion problem. In this case we can fall back on the heuristic search of the state space and search it repeatedly until a desired level of confidence is reached that the bugs are fixed.

V. Conclusion

In conclusion we believe that SBSE and software model checking are complementary techniques that can be leveraged in several different ways. In particular we advocate that SBSE can be used to improve the model checking process and SBSE together with model checking can be used to address common Software Engineering problems. With respect to model checking we have highlighted existing work on an EDA approach to model checking software. We have also described several avenues of future work including leveraging SBSE to optimize the model checking search algorithm to the program under analysis, to a set of domain-specific programs or to a specific bug pattern as well as leveraging SBSE in the context of incremental model checking. With respect to the combined use of SBSE and software model checking to address more general problems we have discussed the use of both techniques in generating test suites as well as our own ongoing research on the automatic repair of concurrency bugs.

VI. Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC). We thank the reviewers for their thoughtful comments and suggestions.

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics*. Springer, 2001, pp. 176–194.

[2] K. Lakhotia, P. McMinn, and M. Harman, "Automated test data generation for coverage: Haven't we solved this problem yet?" in *Proc. of Testing: Academic and Industrial Conference-Practice and Research Techniques, 2009. TAIC PART'09.* IEEE, 2009, pp. 95–104.

[3] M. Harman, "The current state and future of search based software engineering," in *Proc. of Future of Software Engineering*. IEEE, 2007, pp. 342–357.

[4] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc of EEE International Conference on Neural Networks*, vol. 4. IEEE, 1995, pp. 1942–1948.

[5] D. Beasley, R. Martin, and D. Bull, "An overview of genetic algorithms: Part 1. fundamentals," *University computing*, vol. 15, pp. 58–58, 1993.

[6] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys (CSUR)*, vol. 41, no. 4, p. 21, 2009.

[7] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[8] M. Dwyer, J. Hatcliff *et al.*, "Bogor: an extensible and highly-modular software model checking framework," in *ACM SIGSOFT Software Engineering Notes*, vol. 28. ACM, 2003, pp. 267–276.

[9] F. Chicano and E. Alba, "Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models," *Information Processing Letters*, vol. 106, no. 6, pp. 221–231, 2008.

[10] S. Edelkamp, V. Schuppan, D. Bošnački, A. Wijs, A. Fehnker, and H. Aljazzar, "Survey on directed model checking," *Model Checking and Artificial Intelligence*, pp. 65–89, 2009.

[11] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[12] J. Staunton and J. Clark, "Searching for safety violations using estimation of distribution algorithms," in *Proc. of 3rd International Workshop on Search-Based Software Testing (SBST 2010)*, 2010.

[13] ——, "Applications of model reuse when using estimation of distribution algorithms to test concurrent software," *Proc. of 3rd International Symposium on Search Based Software Engineering (SSBSE 2011)*, pp. 97–111, 2011.

[14] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," *Proc. of the 13th International Conference on Software Engineering (ICSE 08)*, pp. 291–300, 2008.

[15] H. J. Goldsby and B. H. Cheng, "Avida-MDE: a digital evolution approach to generating models of adaptive software behavior," in *Proc. of the 10th Annual Conference on Genetic and Evolutionary Computation (GECCO 2008)*. ACM, 2008, pp. 1751–1758.

[16] G. Katz and D. Peled, "Genetic programming and model checking: Synthesizing new mutual exclusion algorithms." Springer, 2008, pp. 33–47.

[17] ——, "Code mutation in verification and automatic code correction." Springer, 2010, pp. 435–450.

[18] W. Visser, C. S. Psreanu, and S. Khurshid, "Test input generation with Java PathFinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[19] A. Hessel and P. Pettersson, "A global algorithm for model-based test suite generation," *Electronic Notes in Theoretical Computer Science*, vol. 190, no. 2, pp. 47–59, 2007.

[20] G. Fraser and F. Wotawa, "Redundancy based test-suite reduction," *Proc. of 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, pp. 291–305, 2007.

[21] ——, "Using LTL rewriting to improve the performance of model-checker based test-case generation," in *Proc. of the 3rd International Workshop on Advances in Model-Based Testing (A-MOST 2007)*. ACM, 2007, pp. 64–74.

[22] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proc. of 34th International Conference on Software Engineering (ICSE 2012)*, Jun. 2012, p. 11.

[23] A. Arcuri, "Evolutionary repair of faulty software," in *Applied Soft. Computing*, vol. 11, 2011, pp. 3494–3514.