Concurrent Bounded Model Checking

Quoc-Sang Phan
Queen Mary University of London, UK
q.phan@qmul.ac.uk

Pasquale Malacaria
Queen Mary University of London, UK
p.malacaria@qmul.ac.uk

Corina S. Păsăreanu
Carnegie Mellon Silicon Valley, NASA Ames, USA
corina.s.pasareanu@nasa.gov

ABSTRACT

We introduce a methodology, based on symbolic execution, for Concurrent Bounded Model Checking. In our approach, we translate a program into a formula in a disjunctive form. This design enables concurrent verification, with a main thread running symbolic execution, without any constraint solving, to build subformulas, and a set of worker threads running a decision procedure for satisfiability checks.

We have implemented this methodology in a tool called JCBMC, the first bounded model checker for Java. JCBMC is built as an extension of Java Pathfinder, an open-source verification platform developed by NASA. JCBMC uses Symbolic PathFinder (SPF) for the symbolic execution, Z3 as the solver and implements concurrency with multi-threading.

For evaluation, we compare JCBMC against SPF and the Bounded Model Checker CBMC. The results of the experiments show that we can achieve significant advantages of performance over these two tools.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Algorithms, Performance, Theory, Verification

Keywords

Bounded Model Checking; Concurrency; Symbolic Execution

1. INTRODUCTION

Bounded Model Checking (BMC) [8] is a popular verification technique that works by unwinding the transition relation of a program for a fixed number of steps k and checking whether a property violation can occur in k or fewer steps. This property checking is then performed by a SAT or SMT solver. BMC is widely used in the hardware industry.

For software, the application of BMC for ANSI-C is embodied in the CBMC tool [12]. In CBMC, a C program containing assertions is encoded into a formula [13] (in Static Single Assignment form), which is then fed to a SAT solver or an SMT solver to check its satisfiability. A satisfying assignment indicates that an error was found. The formula is in conjunctive normal form.

BMC has not been explored so far for many other languages, including Java. However, explicit-state model checking tools such as Java PathFinder (JPF) [2] have been very effective for the verification of many Java applications. Furthermore, there has been an explosion of symbolic execution [22] tools that have been used successfully for many languages [14, 15, 29, 10]. In particular, relevant to our approach, Symbolic PathFinder (SPF) [27] a symbolic execution tool built as an extension of JPF, that provides a symbolic analysis for Java programs.

In this paper, based on our previous work [26], we propose a new methodology for BMC which is based on "classical" symbolic execution (SE) in the sense of King [22]. The way CBMC transforms a program into Static Single Assignment form can also be viewed as executing the program symbolically. However, this encoding is different from the Symbolic Execution of King and we evaluate the two encodings as part of the work reported here. Our methodology is not language specific and only relies on a symbolic executor for that language and an SMT solver.

By using SE we obtain a translation of a program and assertions into a disjunctive formula encoding the path conditions for each bounded (complete) path explored in the code. This suggests a simple concurrent verification strategy that relies on the observations that any subset of disjuncts can be separately checked for satisfiability and whenever a subset is found to be satisfiable the satisfiability task can be stopped. Hence the verification of disjunctive formulas is naturally parallelizable.

We have implemented this methodology in a tool JCBMC, a **J**ava **C**oncurrent **B**ounded **M**odel **C**hecker, which uses SPF to generate the disjunctive formula from the code (constraint solving is turned off in SPF itself) and while generating the formula it sends sub-formulas to multiple worker threads for satisfiability checking. JCBMC handles programs with multi-threading and recursive input data structures and relies on a standard SMT solver, namely Z3 [3], for solving the constraints. Other solvers can be incorporated and different solvers can be used for solving different path constraints in parallel. Although JCBMC is only a prototype, its performance, compared with existing tools, i.e. SPF and CBMC, is remarkable. We summarize our contributions as follows:

- A methodology for concurrent bounded model checking that is based on "classical" SE and it is naturally parallelizable.
- The methodology is language independent and supports assume-guarantee reasoning.
- A tool JCBMC, a concurrent bounded model checker for Java.
- Experiments to show effectiveness of the tool for verification of programs with multi-threading and data structures.
- Comparisons with bounded model checking and "classical" SE, as embodied by CBMC and SPF respectively.

2. BACKGROUND

A program P is modelled as a transition system:

$$P = (S, I, F, T)$$

where S is the set of program states; $I \subseteq S$ is the set of initial states; $F \subseteq S$ is the set of final states; and $T \subseteq S \times S$ is the transition relation. Under this setting, a trace of (a concrete) execution of the program P is represented by a sequence of states:

$$\rho = s_0 s_1 .. s_k$$

such that $s_0 \in I, s_k \in F$ and $\langle s_i, s_{i+1} \rangle \in T$ for all $i \in \{0, .., k-1\}$.

2.1 Bounded Model Checking and CBMC

A trace can be also seen in logical form: the set I and the relation T can be written as their characteristic functions: $s_0 \in I$ iff $I(s_0)$ holds; $\langle s_i, s_{i+1} \rangle \in T$ iff $T(s_i, s_{i+1})$ holds. In this way, a trace ρ is represented by the formula:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

Clearly the transition system P is a model for such a formula, i.e. P is a model for all formulas representing traces of the program. The aim of BMC is to find bugs or prove their absence up to some bounded k number of transitions. That means it explores all traces $\rho = s_0 s_1 .. s_k$ of the program P, in which s_k needs not to be in F. Notice that because of the bound k there are only a finite number of traces to explore hence we can represent the bounded program as a formula $\mathcal{C}$ which is a conjunction of formulas, whose conjoints are possible traces. Notice formulas can also represent symbolic traces, for example if in a formula the value of a program variable is left unspecified then there can be several concrete traces satisfying that formula. Formulas satisfied by set of concrete traces can be referred to as symbolic traces.

CBMC translates a C program into a logical formula $\mathcal{C}$ which is then used as a model for the property $\mathcal{P}$ to be verified. The property is verified by the C program iff $\mathcal{C} \wedge \mathcal{P}$ is valid. This can be checked by a satisfiability solver on $\mathcal{C} \wedge \neg\mathcal{P}$. In fact if $\mathcal{C} \wedge \neg\mathcal{P}$ is true in the model then one trace will satisfy $\neg\mathcal{P}$ hence the property is not valid. On the other hand if $\mathcal{C} \wedge \neg\mathcal{P}$ is false in the model then no trace will satisfy $\neg\mathcal{P}$ hence $\mathcal{P}$ is valid.

2.2 Symbolic PathFinder: Symbolic Execution for Java Bytecode

Symbolic PathFinder (SPF) is a symbolic execution framework built on top of the Java PathFinder (JPF) model checking toolset for Java bytecode analysis. SPF implements a Bytecode interpreter that replaces the standard, concrete execution semantics of bytecodes with a non-standard symbolic execution. Nondeterministic choices in branching conditions are handled by means of JPF's choice generators. Each non-deterministic choice has associated a path condition. JPF's listeners are used to monitor and influence the symbolic execution and to collect and print its results. Symbolic execution of looping programs may result in an infinite symbolic execution tree; for this reason, SPF is run with a user-specified bound on the search depth. By default, whenever the path condition is updated, SPF invokes an off-the-shelf solver to check its satisfiability; if the path condition is found to be unsatisfiable, SPF backtracks. As a result, by default, SPF explores only feasible paths. Note that SPF also has an option to run with no solving (that we use in our work here); as a result of this option, SPF will explore all the possible paths (feasible and infeasible) through the program, up to the given bound. SPF uses *lazy initialization* [20] to handle dynamic input data structures (e.g., lists and trees). Multi-threading is handled systematically using the search mechanisms in JPF core.

3. CONCURRENT BOUNDED MODEL CHECKING

Our method for concurrent bounded model checking is illustrated in Fig. 1. The inputs are: a program under test, a property to verify and three parameters – B is the search bound, N is the number of workers and D is the number of disjuncts to give each worker. The goal is to check if the property holds in the program, up to bound B.

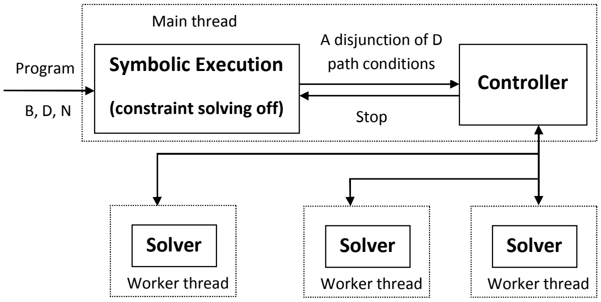

Figure 1: Concurrent BMC architecture

The program under test is analysed using the "classical" (bounded) SE procedure with constraint solving turned off. This means whenever a *path condition* is updated, we do not check its satisfiability, but rather continue the exploration. As a result, SE may explore infeasible paths, which will be checked later using constraint solving. Our approach can be used for the bounded verification of safety properties, which we assume have been reduced to checking assertions embedded in the code. Furthermore, our method supports both *assume* and *assert* statements to enable *assume-guarantee style* verification. The assumed conditions are simply added to the path conditions during the symbolic execution.

The result of SE is a disjunction of path conditions, encoding constraints on the inputs to follow those paths, up to the pre-specified search bound. Among these paths, only the ones that may lead to assert violations are selected for solving. This is achieved by the `Controller` which collects sets of D violating path conditions and sends them for solving to parallel worker threads, using off-the-shelf solvers. The workers start solving as soon as they receive the disjunctive formulas, which may happen while the symbolic execution is still exploring the program. The verification terminates as soon as one of the threads finds a satisfying assignment, in which case an error is reported, or when all the disjunctions are found to be un-satisfiable, in which case the assertion holds (no error) up to the given bound. Note that if the symbolic executor discovers no potentially violating paths (i.e. the error is unreachable), then no solving will be performed.

In general, we use SE to explore all possible symbolic paths up to a certain length, and then encode the program together with the property to check into a formula of the form: $\bigvee_{i=0}^{M} pc_i$ where M is the number of paths that may trigger the error. This form allows us to divide the formula into blocks of D disjunctions:

$$\bigvee_{i=0}^{D-1} pc_i \vee \bigvee_{i=D}^{2D-1} pc_i \cdots \vee \bigvee_{i=(k-1)D}^{kD-1} pc_i \vee \bigvee_{i=kD}^{M} pc_i$$

In this way, we can solve the formula concurrently using several threads, each one solving a single block. A model of a single block is also a model of the formula, therefore the procedure stops when any of the threads find out a model. In JCBMC, after the main thread generates a sub-formula and passes it to a worker thread, it moves on to generate the next sub-formula, while the worker thread solves the given sub-formula concurrently.

3.1 Comparing our approach with Bounded Model Checking and Symbolic Execution

Compared with classical BMC we use an explicit enumeration of paths, while BMC uses an implicit enumeration of paths. Although at first glance the implicit encoding should be better our experiments, even with sequential JCBMC, show that this is not the case. Furthermore, the explicit enumeration is easily parallelizable, with simple and natural load balancing for different threads. Crucially our approach stops as soon as a path leading to an error is found to be satisfiable, while with classical BMC, all the program needs to be explored.

Compared with classical symbolic execution: we solve only in the end. So obviously the price to be paid is the exploration of infeasible paths. On the other hand, again, it is naturally parallelizable and constraint solving, which is one of the bottlenecks in SE, can be done in parallel, even with different solvers, with little coordination, if any, needed.

SPF is used to extract all possible symbolic paths up to a given conditional branching depth. The proposed method outperforms SPF (with solving), even in the sequential version (probably even with one worker). One reason could be that most programs do not contain unreachable symbolic paths, therefore ”lazy” checking of complete symbolic paths is more efficient than checking prefixes after each branching. This hypothesis could be validated by applying SPF such a lazy way or using incremental SMT solving.

The proposed method does not check each single symbolic path separately but it checks sets of paths. This way it makes use of the clause learning mechanism of SMT solving, in contrast to SPF. It would be also interesting to consider the exchange of certain learnt clauses between the workers to further speed up the satisfiability checks.

The proposed method outperforms classical BMC because it uses substitution instead of introducing sets of variables for each (relevant) program location. It would be interesting to see whether a classical BMC formulation checked by an SMT solver whose theory solver applies substitution as pre-processing would yield similar results.

4. EVALUATION

Our evaluation comprises cases studies to compare JCBMC with SPF (v6, default configuration) and case studies to compare JCBMC with CBMC[1]. To compare with CBMC we have considered C code whose Java translation is almost literal. By JSBMC we denote the sequential implementation of JCBMC where a single worker is used. Experiments are run on a machine equipped with dual Xeon(R) E5-2670 CPUs. The results are shown in Tables 2 and 3. Unless otherwise specified times are in seconds, xmy means x minutes and y seconds, “timed out” is one hour and x denotes a memory hit[2]. The source code of JCBMC and the examples can be found at: `https://github.com/qsphan/jpf-bmc`

4.1 Comparing with CBMC and SPF

We evaluate our tool against CBMC and SPF in two classical programs that can be written in both C and Java.

4.1.1 Bubble Sort

We consider the classical bubble sort algorithm, which has already been studied in the BMC community [1, 5]. Here, differently from [5], we consider the more challenging symbolic version where the values of the array are non-deterministically chosen. We consider both the verification of the assertion “the elements of the array are ordered after bubble sort” and its negation “the elements of the array are not ordered after bubble sort”. We analyse a program implementing bubble sort. It will hence contain no bugs for the positive assertion and will be buggy for the negation. Results are shown in Fig 2. We notice that while CBMC is better for the positive assertion, JCBMC outperforms the other tools for the negative assertion and is capable of find a counterexample for array sizes of a higher order of magnitude.

4.1.2 Sum of array

We consider the array case studies from the Software Verification competition 2014 [1], in particular *sum_array_safe.c* for verification and *sum_array_unsafe.c* for refutation. The array size is set to 1000. Results show both SPF and JCBMC outperform CBMC for the unsafe version, while CBMC has a slight advantage for the safe version.

4.2 Comparing with SPF (Java code)

The following examples consist of substantial Java code which is not naturally translatable in C; we hence compare JCBMC only with SPF. Notice JCBMC and SPF are both extensions of JPF: in the case all inputs are concrete they both reduce to JPF-core hence their performance is identical. Hence we only consider programs with symbolic inputs.

4.2.1 Flap controller

This case study is shipped with the distribution of SPF. It is a multi-threaded program modelling a simplified flap controller on an aircraft. It contains 3 classes, and 80 lines of code.

4.2.2 Red Black Tree

This is another example from the SPF distributions (3474 LOC in one class). We check for consistency of the tree after performing `put`, `remove`, `get` and `firstKey` symbolically.

4.2.3 MER Arbiter

The MER Arbiter models a component of the flight software for NASA JPL’s Mars Exploration Rovers (MER) [6]. The MER Arbiter has been modelled in Simulink/Stateflow and it was automatically translated into Java using the Polyglot framework and analyzed with SPF. The configuration for our analysis involved two users and five resources. The example has 268 classes, 553 methods, 4697 lines of code (including the Java Polyglot execution framework) but only approx. 50 classes are relevant. We analyse the code with and without the error (see [6]).

[1] To compare both tools with the same solver in the experiments CBMC will be called with option –smt2, and we will use Z3 for satisfiability checks. Note also that CBMC has an option -z3 to use Z3 in SMT1 format. However, in our case studies, using CBMC with this option is much slower than using with SMT2 as in our experiments.

[2] A memory hit is a “run out of memory” problem. This can be addressed by a different memory manager in JPF.

	SPF	**JSBMC**	**CBMC**	**JCBMC** (D = 10)	**JCBMC** (D = 200)
Array size	Bubble sort with assertion negated				
6	5.622	12.604	0.817	1.160	1.389
30	4m32.790	x	timed out	1.387	2.905
100	timed out	x	timed out	4.944	34.697
	Verification of bubble sort				
5	6m19.222	3.712	7.171	4.193	3.622
6	timed out	26.293	37.816	29.512	21.834
7	x	x	5m22.641	x	x
8	x	x	timed out	x	x
Sum of array					
unsafe	1.403	12.671	1m5.738	1.576	2.479
safe	failed	12.030	2.252	9.466	10.614

Figure 2: Performance of all tools. "failed" refers to SPF failing to solve the constraints using the integrated solver.

Tool	**SPF**	**JSBMC**	**JCBMC** (D = 10)	**JCBMC** (D = 200)
Flap controller (unsafe)	1.141	2.899	0.948	1.370
Red-black tree (safe)	53.602	3.942	3.267	2.774
MER Arbiter (unsafe)	5.275	8.111	7.479	7.579
MER Arbiter (safe)	47.065	59.145	57.740	58.886

Figure 3: Performance on Flap controller, Red-black tree and MER Arbiter

4.2.4 Discussion

Compared to CBMC, JCBMC scores better in finding counterexamples than in verifying their absence; this is consistent with its design because a counterexample corresponds to a worker thread finding a model of the formula. Compared with SPF, JCBMC can be much better (see bubble sort or red black tree) but can also be comparable or slightly worse (see MER Arbiter and Flap Controller results). The reason for the latter is that the cost of generating path conditions dominates the cost of solving them. Similarly, SPF failed to generate formulas for bubble sort for sizes 7 and higher. Furthermore, an error path (e.g. in MER) may occur at the beginning of the SE exploration, and it is therefore discovered quickly by SPF, while JCBMC still needs to generate the pre-specified number D of error paths before solving them. The results suggest one direction for future work, namely to investigate improving the cost of SE-based path generation.

5. RELATED WORK

Related approaches on parallelising BMC [4, 34] address parallel solving of the conjunctive formula that is built for BMC and aim at performing solving at different bounds, where some clauses are shared to enable more efficient SAT solving. In contrast we aim to solve the formulas generated with SE for the same bound, which are naturally disjoint resulting in a simpler parallel algorithm. Furthermore our work aims at verifying programs written in high-level languages such as Java and it is not clear how the previous work, performed in the context of finite state automata, would be applicable. Also related is the work on parallel SAT and SMT solving [31, 28, 35], which can be seen complementing our work; we can use, for example, the parallel version of Z3 [35] in each of the workers to further speed up our proposed approach.

PKIND [19] is a parallel model checker for Lustre that uses k-induction. PKIND runs in parallel the different tasks involved in performing the induction: the base step, the induction step and also the generation of auxiliary invariants used for verification. Thus it performs the parallel work at a higher level of granularity than JCBMC. It would be interesting to investigate if we can replace the parallel tasks in PKIND with our own version of SE-based bounded verification, which in turn is parallelized at the level of granularity of symbolic paths. Parallel model checking has been investigated in the context of explicit-state [7, 25, 9, 17, 33, 18] and symbolic [24, 23] exploration. The latter were done in the context of Binary Decision Diagrams, and hence are very different from ours. These approaches concentrate on partitioning the state space to be explored in parallel and on dealing with the communication overhead between parallel workers. In contrast, in our approach the workers perform the solving independently, with no communication between them.

Previous work on parallel symbolic execution include [32, 21, 11, 30, 16]. All these approaches were done in the context of "classical" or dynamic symbolic execution, using constraint solving during path generation. In contrast the approach we advocate here has a clear separation between path generation and constraint solving, allowing us to easily achieve load balancing between workers, with little communication overhead.

6. CONCLUSION AND FUTURE WORK

We have presented a language independent methodology for concurrent BMC. Based on this methodology we have implemented a concurrent bounded model checker for Java. Future work includes to upgrade its concurrency from single CPU multi-threading to true parallelism and to perform obvious optimisations like replacing SPF with a lighter weight tool, or a parallel version, to reduce the cost of generating path conditions. It will also be interesting to implement the methodology for other languages (C, Python) and investigate how to use SE for IC3 style verification.

7. ACKNOWLEDGEMENTS

We thank Michael Tautschnig for his valuable comments on an earlier version of this paper.

8. REFERENCES

[1] Benchmarks of loops in the Software Verification 2014 competition 2014. `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp14/loops/`.

[2] Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf/`.

[3] Z3. `http://z3.codeplex.com/`.

[4] Erika Ábrahám, Tobias Schubert, Bernd Becker, Martin Fränzle, and Christian Herde. Parallel sat solving in bounded model checking. FMICS'06/PDMC'06, pages 301–315.

[5] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *STTT*, 11(1):69–83, January 2009.

[6] Daniel Balasubramanian, Corina S. Păsăreanu, Gábor Karsai, and Michael R. Lowry. Polyglot: systematic analysis for multiple statechart formalisms. TACAS'13, pages 523–529.

[7] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *HiBi/PDMC 2010*, pages 4–7. IEEE, 2010.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[9] Ethan Burns and Rong Zhou. Parallel model checking using abstraction. SPIN'12, pages 172–190, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, pages 209–224.

[11] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *SIGOPS Oper. Syst. Rev.*, 43(4):5–10, January 2010.

[12] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[13] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of c and verilog programs using bounded model checking. In *Proceedings of the 40th annual Design Automation Conference*, DAC '03, pages 368–371, New York, NY, USA, 2003. ACM.

[14] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. PLDI '05, pages 213–223. ACM, 2005.

[15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[16] Elsa L. Gunter and Doron Peled. Unit checking: Symbolic model checking for a unit of code. In Nachum Dershowitz, editor, *Verification: Theory and Practice, Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 548–567. Springer, 2003.

[17] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Trans. Softw. Eng.*, 33(10):659–674, October 2007.

[18] Shahid Jabbar and Stefan Edelkamp. Parallel external directed model checking with linear i/o. VMCAI'06, pages 237–251, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] Temesghen Kahsai and Cesare Tinelli. Pkind: A parallel k-induction based model checker. In Jiri Barnat and Keijo Heljanko, editors, *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.

[20] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. TACAS'03, pages 553–568. Springer-Verlag, 2003.

[21] Andrew King. Distributed parallel symbolic execution. In *Master Thesis, Kansas State University*, 2009.

[22] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[23] Marta Z. Kwiatkowska, Alessio Lomuscio, and Hongyang Qu. Parallel model checking for temporal epistemic logic. In *ECAI*, pages 543–548, 2010.

[24] Pradeep K. Nalla, J. Weiss, JÃijrgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel. Parallel bounded property checking with symc.

[25] Robert Palmer and Ganesh Gopalakrishnan. Partial order reduction assisted parallel modelchecking (full version. Technical report, PDMC'2002, 2002.

[26] Quoc-Sang Phan. Symbolic execution as dpll modulo theories. In *2014 Imperial College Computing Student Workshop*, volume 43 of *OpenAccess Series in Informatics (OASIcs)*, pages 58–65, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[27] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 179–180, New York, NY, USA, 2010. ACM.

[28] T. Schubert, M. Lewis, and B. Becker. Pamira - a parallel sat solver with knowledge sharing. In *MTV '05*, pages 29–36, 2005.

[29] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. ESEC/FSE-13, pages 263–272. ACM, 2005.

[30] J.H. Siddiqui and S Khurshid. Parsym: Parallel symbolic execution. In *ICSTE*, volume 1, pages V1–405–V1–409, 2010.

[31] Carsten Sinz, Wolfgang Blochinger, and Wolfgang KÃijchlin. Pasat - parallel sat-checking with lemma exchange: Implementation and applications. In *SAT*, 2001.

[32] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. ISSTA '10, pages 183–194, New York, NY, USA, 2010. ACM.

[33] Ulrich Stern and David L. Dill. Parallelizing the murphi verifier. CAV '97, pages 256–278, London, UK, UK, 1997. Springer-Verlag.

[34] Siert Wieringa, Matti Niemenmaa, and Keijo Heljanko. Tarmo: A framework for parallelized bounded model checking. In *PDMC*, pages 62–76, 2009.

[35] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. A concurrent portfolio approach to smt solving. In *CAV*, pages 715–720, 2009.