Parallel Property Checking with Symbolic Execution

Junye Wen, Guowei Yang
Department of Computer Science
Texas State University, San Marcos, TX
{j_w236, gyang}@txstate.edu

***Abstract*—Systematically checking code against functional correctness properties is costly, especially for complex code annotated with rich behavioral properties. This paper introduces a novel approach to checking properties in parallel using symbolic execution. Our approach partitions a check for the whole set of properties into multiple simpler sub-checks—each sub-check focusing on a single property, so that different properties are checked in parallel among multiple workers. Furthermore, each sub-check is *guided* by the checked property to avoid exploring irrelevant paths and is *prioritized* based on distances towards the checked property to provide early feedback. We implement our approach in Symbolic PathFinder, and experiments on systematically checking assertions in Java programs show the effectiveness of our approach.**

I. Introduction

Researchers have long recognized the value of annotating functional correctness properties of code using assertions [7] or executable contracts, such as those supported by the Java Modeling Language [15] or Eiffel [17]. However, developers are often reluctant to use them largely due to the high computational cost of running automated analyses to check them.

Symbolic execution [13], [14] is a powerful program analysis technique that has a number of useful applications and has been widely used as a systematic technique for bug finding [10], [20], [22], [28], but symbolic execution is computationally expensive due to the large number of paths to explore as well as the high cost of underlying constraint solving. Scaling symbolic execution remains challenging for complex programs in practice. When programs are annotated with functional correctness properties, symbolic execution can be naturally applied to automatically check program behaviors against the annotated properties to check their validity. However, the scalability issue is even exacerbated as the annotated properties often introduce extra paths and extra constraints. This paper is focused on reducing the computational cost of symbolic execution in checking properties.

A lot of advances in symbolic execution have been made during the last decade. Specifically, parallel analysis [5], [23]–[25] allows multiple workers to explore largely disjoint sets of program behaviors in parallel, and has shown particular promise in addressing the scalability issue of symbolic execution. However, to the best of our knowledge, none of the approaches consider the characteristics of the annotated properties in their parallelization strategies.

DOI reference number: 10.18293/SEKE2018-171

This paper introduces a novel approach to parallel property checking using symbolic execution. Our key insight is that properties are normally written without side effects, and thus checking of each property is independent of checking of other properties. Our approach partitions a check for the whole set of properties into multiple simpler sub-checks—each focusing on one single property, so that different properties are checked in parallel among multiple workers. Furthermore, each sub-check is *guided* by the checked property to avoid exploring irrelevant paths and is *prioritized* based on distances towards the checked property to provide earlier feedback, allowing users to fix bugs in code or refine properties earlier. Specifically, during state space exploration we statically check whether the checked property is reachable or not along the current path, and prune the search when the checked property cannot be reached. Moreover, we prioritize the state space exploration so that the state whose corresponding location has the shortest distance towards the checked property is explored first, i.e., the shortest path to the checked property gets explored first. Therefore, the prioritized state space exploration can provide earlier feedback on the checked property. Note that the chance of pruning irrelevant state space is much higher in each sub-check than in the original check, since in a sub-check the program under analysis has only one property at a particular location in the program, while the program under analysis in the original check has multiple properties scattered in different locations in the program.

We implement our approach in Symbolic PathFinder [18]. To evaluate the efficacy of our approach we apply it in the context of symbolic execution for checking Java programs annotated with assertions. We conduct experiments based on five subjects: three Java programs with manually written assertions and two Java programs with synthesized assertions. Experimental results show that our approach for parallel property checking detects more assertion violations and reduces the overall analysis time compared with regular non-parallel property checking. For one subject, while regular property checking timed out after executing for two hours, our parallel property checking technique completed within four seconds. In addition, for most sub-checks, our guided check prunes state space and reduces the time cost, and our prioritized check provides earlier feedback compared to regular check.

II. Motivating Example

We use an example to illustrate how our approach leverages the annotated properties to improve the scalability of symbolic execution for property checking. Consider the source code of `median` shown in Figure 1. It computes the middle value of its three integer inputs; this method is adapted from previous work [12], and five assertions are manually added to check the correctness of the program. For example, the user asserts $x <= y$ && $y <= z$ at line 4, indicating that y should be the middle value of the three inputs; otherwise, an assertion violation is captured.

```
1 int median(int x, int y, int z) {
2     if (y < z) {
3         if (x < y){
4             assert x <= y && y <= z; //#1
5             return y;}
6         else if (x < z){
7             assert y <= x && x <= z; //#2
8             return x;}
9     }
10    else {
11        if (x > y){
12            assert z <= y && y <= x; //#3
13            return y;}
14        else if (x > z){
15            assert z <= x && x <= y; //#4
16            return x;}
17    }
18
19    assert (x<=z && z<=y) || (y<=z && z<=x); //#5
20    return z;
21 }
```

Fig. 1. Method to compute the middle value of three input numbers and its annotated assertions.

The workload of checking five assertions in this program is conducted by five workers running in parallel, such that each worker checks one single assertion. For example, the worker responsible for checking assertion #1 analyzes a program version, where the code together with the target assertion #1 remain unchanged, while all the other four assertions are removed.

In addition, each sub-check is further optimized using *guided* and *prioritized* state space exploration based on the checked assertion. For checking assertion #1, the sub-check is guided by assertion #1, avoiding exploring the irrelevant parts of the program. Therefore, instead of exploring all the six possible paths in the program, the guided check only explores one path, that satisfies path condition $y < z$ and $x < y$ and reaches the checked assertion. It results into up to 5/6 reduction in terms of the number of paths to be explored. If multiple paths can reach the checked assertion, we use shortest distance based heuristics to prioritize the search so that the assertion can be checked as early as possible and a feedback, i.e., whether the assertion is violated or not, can be returned to the user as early as possible.

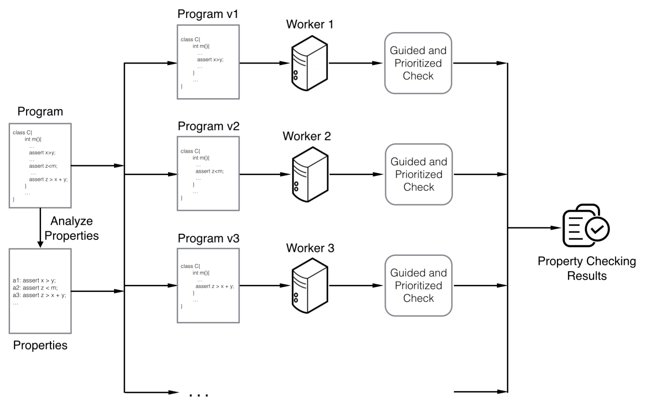

Fig. 2. An overview of the approach

III. Approach

Our approach is focused on how to optimally utilize the computing resources available to check properties, specifically in a parallel setting where the checking can be conducted among several workers. Our key insight is that properties are normally written without side effects, and thus checking of each property is independent of checking of other properties. The result from checking all properties in one run should be the same as that from checking properties in multiple multiple runs in parallel. This enables us to partition a check for the whole set of properties into multiple simpler checks–each focusing on one single property, so that different properties are checked in parallel among multiple workers. Therefore, the original check is converted into multiple simpler sub-checks in parallel for better scalability.

Figure 2 shows an overview of the approach. Consider a program P with multiple properties $PT = \{PT_1, PT_2, ..., PT_m\}$ to check. Our approach first statically analyzes the program to find all the m properties to check, and accordingly prepare m program versions $V = \{v_1, v_2, ..., v_m\}$ where each version contains only one property that does not appear in other versions. These versions are then checked by m workers, each worker focusing on one version and altogether checking all the properties in parallel. Each worker works on its own program version with one single property using property guided and prioritized check. Finally, the property checking results from these workers are delivered to the user.

The partition of properties not only simplifies the program to be checked due to the removal of other properties, but also allows further optimization of each sub-check. Since each sub-check focuses on one single property, it is more likely to have paths that do not reach the checked property compared with the original check that focuses on multiple properties. Leveraging this observation, each sub-check, i.e., a symbolic execution run for checking one single property, is *guided* by the checked property such that it only explores the program state space that is relevant to the checked property. If the current path cannot reach the checked property, symbolic execution does not continue along the path and backtracks. By effectively

Algorithm 1 Procedure *check* for checking a property

```
Input: Program $P$, property $PT$, search depth bound $DepthBound$
Output: A set of property violations detected during symbolic execution $VS$

1: Queue $tq \leftarrow$ enabled transitions at current state $s$
2: while $\neg tq.isEmpty()$ do
3:     $t \leftarrow tq.remove()$
4:     $nt \leftarrow GetCFGNode(P, t)$
5:     $na \leftarrow GetCFGNode(P, PT)$
6:     if $\neg IsCFGPath(nt, na)$ then
7:         continue
8:     else
9:         $s' \leftarrow execute(s, t)$
10:        $pc \leftarrow$ current path condition
11:        $depth \leftarrow depth + 1$
12:        if $pc$ is not satisfiable then
13:            continue
14:        end if
15:        if $isPropertyViolated(s')$ then
16:            $VS.add(violation(s'))$
17:            continue
18:        end if
19:        if $depth == DepthBound$ then
20:            continue
21:        else
22:            $check(s')$
23:        end if
24:    end if
25: end while
```

pruning paths that cannot reach the checked assertions, our approach avoids the cost of exploring irrelevant paths.

Algorithm 1 shows the procedure *check* for performing property checking for a program with one single property. Given as input a program, a property to check, and a bound on the search depth, the procedure check the conformance of the program behaviors with the checked property, and return all property violations in the program. It starts with the initial state for s, 0 for *depth*, and an empty set for VS. It finds all enabled transitions at the current state (Line 1) to systematically search the state space. Lines 4 – 5 locate the Control Flow Graph (CFG) nodes for the enabled transition, and the checked property, respectively. Both the enabled transition and the checked property could correspond to multiple CFG nodes, we simplify it here assuming that each corresponds to one CFG node. It checks whether the current transition reaches the checked property, and if not prune the search (Lines 6 – 7); otherwise, it executes the transition to get to the next state, and update the pc and depth (Lines 9 – 11). If pc is unsatisfiable (i.e., the corresponding path is infeasible), the checked property is violated, or search depth reaches the bound, it backtracks to explore other un-explored enabled transitions (Lines 12 – 20); otherwise, it recursively explores the states rooted at the new state s' (Line 22).

In addition, each sub-check is *prioritized* to provide early feedback to the user. In the context of property checking, usually one property violation is enough for investigating the violation, and there is no need to find all property violations. Our insight is that the earlier a property is checked, the earlier the user could start the investigation and fix the potential problem either by modifying the code or by refining the checked property. As there is no precise way to predict the feasibility of paths and how long each path would take. We use a heuristics to prioritize the check. Specifically, we calculate the distances from the current point towards the checked property along all potential paths, and choose the shortest path to explore first [16].

To prioritize the search, at each branching point, we sort the list of enabled transitions based on an estimated distance to the checked property in a CFG. For each enabled transition t_i, we compute an estimated distance to the checked property. The enabled transitions queue (*tq* in Algorithm 1) is sorted in ascending order based on the estimated distances of the transitions before the queue is explored. The enabled transition with the shortest distance is explored first. The distance is a lower bound on the number of CFG branches from a node n_i (corresponding to t_i) to node n_j, that is corresponding to the checked property:

$$\forall n_i \ . \ n_j : d_i := min \ (\ branches \ (\ n_i, n_j \ ))$$

In our approach, we use the all-pairs shortest path algorithm to compute the lower bound on the number of CFG branches. The complexity is cubic in the number of branches in the CFG. We note that metrics other than number of branches can also be used as a distance estimate, for example, the number of bytecodes.

IV. Evaluation

We empirically evaluate the effectiveness of our approach for parallel property checking. Our evaluation addresses the following research questions:

◦ **RQ1**: How does the efficiency of our parallel property checking compare with regular property checking?

◦ **RQ2**: How does the cost of our guided check compare with regular check?

◦ **RQ3**: How does our prioritized check compare with regular check in terms of providing feedback to the user?

A. Artifacts

In our evaluation, we use five subjects including `median`, `testLoop`, `trityp`, `WBS`, and `TCAS`. All of them have been used before for evaluating symbolic execution techniques [19], [25], [27], [28].

The first subject `median` is shown in Figure 1. The second subject `testLoop` is used to investigate how our approach can help deal with loops, as they pose particular challenges to symbolic execution and handling them efficiently is an active area of research. The third subject is a Java version of the classic triangle classification program by Ammann and Offutt. The classification logic of the `trityp` program seems deceptively simple, but are non-trivial to reason about. We consider the correct version of assertions developed for `trityp` in previous work [27].

For the two subject programs `WBS` and `TCAS`, we use mechanically synthesized assertions. To synthesize assertions for our experiments, we use the *Daikon* tool for invariant discovery [9]. Specifically, we apply Daikon on each subject to discover invariants and transform them to assertions. Daikon requires a test suite to execute the program under analysis and detect its likely invariants. TCAS had a test suite available in the Software Infrastructure Repository [1], so we used this test suite which contains 1608 tests. For WBS, we wrote a random test generator to create a test suite with 1000 tests. We selected all the eight Daikon invariants for TCAS and randomly selected 25 out of 35 invariants for synthesizing assertions.

B. Experiment Setup

In this work, we use Symbolic PathFinder (SPF) [18], an open-source tool for symbolic execution of Java programs built on top of the Java PathFinder (JPF) model checker [26] to perform symbolic execution. We implemented guided and prioritized check in SPF as a customized listeners, and we built customized control flow graphs to compute estimated distances and reachability information to guide and prioritize property checking. We also conduct experiments using regular symbolic execution as implemented in SPF for comparison. Choco constraint solver [2] is used for solving path conditions involved in symbolic execution.

To evaluate RQ1 and RQ2, symbolic execution is configured to detect all assertion violations; while to evaluate RQ3, symbolic execution is configured to stop when it detects the first assertion violation, to check whether our prioritized check could provide earlier feedback than regular check.

We assume that there are enough workers available for performing the tasks in parallel. In practice, resources could be limited, and we need design strategies for statically grouping work before dispatching or for dynamically stealing work among workers, which is left for future work.

We perform the experiments on the Lonestar cluster at the Texas Advanced Computing Center (TACC) [3]. TACC provides powerful computation nodes with reliable and fast connectivity. The programs for each worker node are executed on independent processors without memory sharing.

C. Results and Analysis

In this section, we present the results of our experiments, and analyze the results with respect to our three research questions.

RQ1: How does the efficiency of our parallel property checking compare with regular property checking?

Table I shows the experimental results for checking all assertions in the subject programs using our parallel property checking approach and using regular non-parallel property checking approach. It shows the number of detected assertion violations, and three types of checking cost, i.e., time, number of states explored, and the maximum memory cost, for each approach. Since in the parallel property checking sub-checks are analyzed in parallel among multiple workers, the table shows cost ranges of values across all sub-checks, and it also shows the overall time cost for the parallel property checking; while for regular symbolic execution the cost is collected by running regular symbolic execution on the original program annotated with all assertions. We note that 0 in time cost means less than 1 second. TO indicates that the corresponding check timed out.

We find that there are no assertion violations for median and trityp, while for the other three subjects, the parallel approach detects more assertion violations than regular approach. This is because some expensive assertion checking happens only in the parallel property checking. Since Symbolic PathFinder backtracks as soon as it detects an assertion violation, the inputs reaching deep assertions may be reduced due to violations of the shallow assertions along the same path, and thus may not detect the possible violations of the deep assertions in regular property checking approach.

Moreover, for all subjects except for WBS, the parallel approach is more efficient than regular approach in property checking. Specifically, it achieves almost $3X$ speedup for TCAS. For testLoop, while regular symbolic execution timed out after executing for *two* hours, the parallel property checking completed within 31 seconds. Without surprise, most sub-checks explored only part of the state space. We also note however for WBS our approach took more time, and explored more states, which is because of the cost for detecting the 130 more violations.

In addition, we find that although the parallel approach takes almost the same memory cost as regular symbolic execution for most runs, it takes more memory for some sub-checks for WBS and TCAS; we note however that the maximum memory reported by SPF may vary a lot due to the underlying garbage collection, and thus this comparison is not very meaningful.

RQ2: How does the cost of our guided check compare with regular check?

Table II reports the experimental results for each sub-check using guided check and prioritized check compared to using regular check, i.e., regular symbolic execution. As we explained before, the comparison in memory cost is not very meaningful, thus here we only report the cost in terms of time and explored states.

To evaluate RQ2, symbolic execution is configured to check for all assertion violations. We observe that for 44 out of 50 versions, our guided check explored fewer states than regular check, since guided check prunes state space exploration when the checked property is not reachable. For example, for $v1$ of testLoop, guided check explored 103 states while regular check explored 154 states, which is about 1/3 reduction. Accordingly, the guided check took less time than regular check for most of these cases. For example, for $v1$ of trityp, guided check took 18 seconds while regular check took 22 seconds. However, we note that for some cases, although there was a reduction in states, the time cost of guided check was even higher than regular check due to the overhead of static analysis involved in guided check.

RQ3: How does our prioritized check compare with regular check in terms of providing feedback to the user?

To evaluate RQ3, run symbolic execution is configured to stop when it finds the first assertion violation. From Table II, we observe that for 40 out of 50 versions, prioritized check

TABLE I
Results of parallel and regular property checking.

Subject	Parallel Property Checking					Regular Property Checking			
	Detected Violations	Total Time (s)	Time (s)	# of States	Memory (MB)	Detected Violations	Time (s)	# of States	Memory (MB)
median (5 assertions)	0	2	0-2	5-13	965-965	0	2	13	965
testLoop (2 assertions)	2	31	0-30	103-180	965-965	-	TO	-	-
trityp (10 assertions)	0	49	18-48	33-49	965-965	0	103	81	965
WBS (8 assertions)	222	7	0-7	359-671	965-1178	92	2	533	965
TCAS (25 assertions)	251	680	27-679	679-935	965-1685	195	2025	2047	965

TABLE II
Property checking using guided and prioritized check and regular check.

Subject	Ver	Check all violations				Check first violation			
		Guided Check		Regular Check		Prioritized Check		Regular Check	
		Time	States	Time	States	Time	States	Time	States
median	v1	0	5	0	11	0	5	0	11
	v2	0	7	1	11	0	7	1	11
	v3	0	5	1	11	0	5	1	11
	v4	1	5	0	11	0	5	1	11
	v5	2	13	2	13	1	13	1	13
testLoop	v1	0	103	0	154	0	103	0	154
	v2	30	727	TO	TO	0	180	TO	TO
trityp	v1	18	33	22	40	18	33	22	40
	v2	18	35	18	36	18	35	21	36
	v3	33	49	40	57	34	49	35	57
	v4	29	39	29	39	29	39	32	39
	v5	48	35	53	36	45	35	55	36
	v6	28	37	30	42	28	37	29	42
	v7	26	37	27	40	26	37	29	40
	v8	19	35	20	36	21	35	21	36
	v9	22	39	19	42	22	39	20	39
	v10	20	39	23	39	23	39	21	39
WBS	v1	0	451	0	455	0	163	0	255
	v2	0	359	0	359	0	222	0	341
	v3	0	527	1	530	0	527	0	530
	v4	0	623	1	623	0	9	0	9
	v5	0	535	1	535	0	535	0	561
	v6	7	671	11	680	0	9	0	9
	v7	0	487	1	500	0	48	0	117
	v8	0	527	0	530	0	368	0	421
TCAS	v1	219	727	250	760	289	727	265	702
	v2	27	727	27	760	37	727	43	702
	v3	34	687	33	702	37	687	33	702
	v4	149	687	156	702	125	687	137	702
	v5	30	679	41	754	27	679	34	754
	v6	35	679	40	754	33	679	37	754
	v7	31	679	35	679	33	679	34	679
	v8	28	679	33	679	33	679	34	679
	v9	241	695	275	722	240	695	257	722
	v10	251	695	270	722	222	695	318	722
	v11	32	695	36	722	1	33	1	38
	v12	31	695	33	722	1	33	1	38
	v13	201	695	226	727	238	695	241	727
	v14	130	695	132	727	134	695	146	727
	v15	28	695	34	727	11	229	13	270
	v16	28	695	35	727	9	229	12	270
	v17	679	743	644	745	557	743	568	745
	v18	31	743	32	745	31	743	32	745
	v19	36	935	34	950	15	370	26	439
	v20	33	935	36	950	8	247	14	323
	v21	30	719	30	874	29	678	29	678
	v22	34	719	35	874	11	167	18	214
	v23	33	815	35	827	0	20	0	33
	v24	28	815	39	827	10	191	20	331
	v25	34	815	35	827	9	211	19	231

explored fewer states than regular check, and for 8 versions, both techniques explored the same number of states. For instance, for *v*24 of `TCAS`, prioritized check explored 191 states, while regular check explored 331 states. However, for the other 2 versions (i.e., *v*1 and *v*2 of `TCAS`, prioritized check explored slightly more states than regular check. This is not surprising as the shortest path selected by our heuristics is based on number of branches in CFG, and may result in more states to explore in symbolic execution. Similar to previous experiments, prioritized check usually took less time when it explored fewer states, as the time cost is correlated with states exploration. For example, for *v*10 of `TCAS`, prioritized check took 222 seconds, while regular check took 318 seconds, which is about 1.5*X* speedup. Moreover, for *v*2 of `testLoop`, prioritized check completes in less than one second; in contrast, regular check timed out after running for *two* hours. Only for few versions, prioritized check took slightly more time than regular check.

V. Related Work

Several research projects have proposed techniques for parallel symbolic execution [5], [23], [25]. Static partitioning [25] leverages an initial shallow symbolic execution run to minimize the communication overhead during parallel symbolic execution. It creates pre-conditions using conjunctions of clauses on path conditions encountered during the shallow run, and restricts symbolic execution by each worker to explore only paths that satisfy the pre-condition. ParSym [23] parallelizes symbolic execution dynamically by taking each path exploration as one unit of work and using a central server to distribute work between parallel workers. Cloud9 [5] utilizes load balancing that initially assigns the whole program analysis to a worker, and whenever an idle worker becomes available, the load balancer instructs the busy worker to suspend exploration and breaks off some of its unexplored subtree to send to the idle worker to balance the work load. While these techniques use parallelization to speed up symbolic execution in general and check the whole bounded state space, our work is focused on checking side-effect-free properties and ignores path exploration that is irrelevant to the checked properties.

Much work has been done for guiding symbolic execution [16], [19], [21]. Directed symbolic execution [19] uses a def-use analysis to compute change affected locations and then uses this information to guide symbolic execution to explore only program paths that are affected by the changes. Santelices and Harrold [21] use control and data dependencies to symbolically execute groups of paths, rather than individual paths. Ma et al. [16] propose a call chain backward search heuristic to find a feasible path to the target location. Our work leverages reachability of properties to guide symbolic execution to only explore paths relevant to the checked properties.

Some recent projects [11], [27], [29] have explored more efficient checking of properties. Guo et al. [11] introduce assertion guided symbolic execution for eliminating redundant executions in multi-threaded programs to reduce the overall computational cost. An execution is considered redundant when it shares the same reason why it cannot reach the bad state with previous executions, and thus can be eliminated for the purpose of checking assertions. While it focuses on eliminating redundant executions for multi-threaded programs, our guided check focuses on eliminating irrelevant executions for single-threaded programs. iProperty [27] computes differences between assertions of related programs in a manner that facilitates more efficient incremental checking of conformance of programs to properties. Our approach is orthogonal and can use iProperty to compute differences between assertion versions when the checked assertion is changed, thus speeding up the assertion checking carried out by each worker. iDiscovery [29] uses assertion separation to focus symbolic execution on checking one assertion at a time, and violation restriction to generate at most one violation of each assertion. While our work shares some insight with assertion separation on checking assertions separately, the guided and prioritized check in our work has potential to more efficiently check each assertion.

This work is different from property-based slicing and property-aware testing and verification [4], [6], [8], since here we simply check the reachability of properties and apply this for guiding symbolic execution rather than other testing or verification techniques.

We have presented the high-level ideas of this work in Java PathFinder workshop 2015 to get early feedback,with no formal proceedings for the paper. In this paper we have developed the ideas further, and we have also provided more evaluation of the work.

VI. Conclusions and Future Work

This paper introduced a novel approach for partitioning the problem of property checking using symbolic execution into simpler sub-checks where each check is focused on checking one single property. All sub-checks are performed by multiple workers in parallel for better scalability. The parallelized property checking enabled us to further optimize each sub-check by pruning irrelevant paths regarding the checked property. Moreover, check is prioritized to explore shorter paths towards properties so that earlier feedback on the checked property can be provided to the user. Experiments using five subject programs with assertions that are manually written as well as automatically synthesized, showed that our approach for parallel property checking reduced the overall analysis time compared with regular non-parallel property checking; and in sub-checks which focus on checking one single assertion, our guided check pruned state space exploration and thus reduced the time cost, and our prioritized check provided earlier feedback compared to regular check.

As for future work, we plan to conduct more extensive evaluation of our approach using more complex subjects, such as open source programs. We would also like to investigate how to parallelize property checking when not enough resources are available, for example, the number of available workers is fewer than the number of checked properties in the program.

Acknowledgments

This work is partially supported by the National Science Foundation under Grant No. CCF-1464123.

References

[1] SIR Repository. http://sir.unl.edu.

[2] Choco solver. http://www.emn.fr/z-info/choco-solver.

[3] Lonestar cluster. https://www.tacc.utexas.edu/systems/lonestar.

[4] R. H. Bordini, M. Fisher, M. Wooldridge, and W. Visser. Property-based slicing for agent verification. *J. Log. and Comput.*, 19(6):1385–1425, Dec. 2009.

[5] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *EuroSys*, pages 183–198, 2011.

[6] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Information & Software Technology*, pages 595–607, 1998.

[7] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Software Engineering Notes*, 2006.

[8] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking java programs. In *ICSE*, pages 762–765, 2000.

[9] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[10] P. Godefroid, S. K. Lahiri, and C. Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*, pages 112–128, 2011.

[11] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE*, pages 854–865, 2015.

[12] J. A. Jones. *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 2008.

[13] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[14] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

[15] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of jml accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, pages 185–208, 2005.

[16] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks. Directed symbolic execution. In *SAS*, pages 95–111, 2011.

[17] B. Meyer, J.-M. Nerson, and M. Matsuo. Eiffel: Object-oriented design for software engineering. In *ESEC*, pages 221–229, 1987.

[18] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic Pathfinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering*, pages 391–425, 2013.

[19] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *PLDI*, pages 504–515, 2011.

[20] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–26, 2008.

[21] R. Santelices and M. J. Harrold. Exploiting program dependencies for scalable multiple-path symbolic execution. In *ISSTA*, pages 195–206, 2010.

[22] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[23] J. H. Siddiqui and S. Khurshid. ParSym: Parallel symbolic execution. In *ICSE*, pages V1–405 – V1–409, 2010.

[24] J. H. Siddiqui and S. Khurshid. Scaling symbolic execution using ranged analysis. In *OOPSLA*, pages 523–536, 2012.

[25] M. Staats and C. Păsăreanu. Parallel symbolic execution for structural test generation. In *ISSTA*, pages 183–194, 2010.

[26] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, pages 203–232, 2003.

[27] G. Yang, S. Khurshid, S. Person, and N. Rungta. Property differencing for incremental checking. In *ICSE*, pages 1059–1070, 2014.

[28] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *ISSTA*, pages 144–154, 2012.

[29] L. Zhang, G. Yang, N. Rungta, S. Person, and S. Khurshid. Feedback-driven dynamic invariant discovery. In *ISSTA*, pages 362–372, 2014.