Verifying and Testing Concurrent Programs using Constraint Solver based Approaches

Dhriti Khanna
IIIT-Delhi
Delhi, India
dhritik@iiitd.ac.in

Rahul Purandare
IIIT-Delhi
Delhi, India
purandare@iiitd.ac.in

Subodh Sharma
IIT Delhi
Delhi, India
svs@cse.iitd.ac.in

***Abstract*—The success of dynamic verification techniques for confirming the absence of bugs in concurrent programs rests on their ability to systematically address the interleaving space arising because of the nondeterminism. However, existing dynamic verification engines suffer from the problem of scalability due to the size of the reachable state space that grows exponentially as the number of parallel entities increases. The second front on which the dynamic verification technique struggles is the dependence on the test cases to drive the program, thus being as efficient as the quality of the test cases. Lastly, any verification technique suffers from the lack of a significant benchmark of bugs to prove its worth. This work tries to improve the area of dynamic verification concerning the limitations as mentioned above. We utilize the worthiness and popularity of constraint solvers and establish our work in the realm of concurrent programs.**

***Index Terms*—concurrency, deadlock detection, test generation, communication deadlocks**

I. Introduction and Motivation

With the advancements in multi-core hardware technology and distributed systems, parallel programming applications have become an essential requisite to manipulate large data and to facilitate multi-tasking of jobs. Ensuring the correctness of these software is a challenging task due to nondeterminism which is a common spirit amongst the applications of multiple parallel programming paradigms. On one extreme, we have formal verification techniques like theorem proving and model-checking which are not scalable, and on the other extreme, we have testing techniques that can not be used to guarantee a lack of errors. Dynamic verification is a lucrative trade-off between testing and formal verification in terms of coverage and scalability. However, we observe the following shortcomings:

1) it requires repeated executions of the application (multiple restarts) to explore the state-space corresponding to an input and, thus, does not scale for large programs,
2) it explores the schedule space corresponding to a concrete test case and is as good as the provided test cases,
3) the assessment of the bug detection tools is limited by the small benchmark data sets.

Explicit-state dynamic verifiers can exhaustively analyze concurrent programs for the absence of concurrency bugs such as deadlocks and assertion violations under a fixed input. However, they require the applications to be run repeatedly to explore all interleavings. New promising techniques like *predictive analysis* that make use of constraint-solving do address the problem of program re-runs. Still, they alone can not be applied to the programs where the shared or communicated data between multiple executing entities influence control-flow decisions (*multi-path* executions). The first part of our research aims to leverage predictive analysis to garner the benefits of dynamic verification but improving on the scalability front, as mentioned in our first observation.

Our second observation states that even though dynamic verification techniques provide guarantees about the absence of errors for a specific execution, their effectiveness depends entirely on the quality of the test cases. The second part of our research work is a step in this direction. We improve the effectiveness of dynamic verification by synthesizing relevant test cases. The requirement of automated test synthesis techniques is orthogonal to the detection techniques and can be used as the first phase in dynamic verification tools.

Bug detection is a well-researched area that has given many static, dynamic, and hybrid tools and techniques to detect concurrency errors. However, the bug detection tools are evaluated for their worth on a limited set of benchmarks gathering which requires a lot of effort and time. Moreover, the value of any individual dataset drops over time as tools adapt to it. Hence, there is a need to automatically synthesize bugs in the code to eradicate the problems mentioned above. We propose this work as the third improvement in the process of creating reliable and worthwhile dynamic verification tools.

Hence, we focus on providing solutions to these interesting, hard, and impacting problems:

1) *Can we combine constraint-solving and dynamic verification to make the latter more scalable to prove the absence of deadlocks in concurrent programs?*
2) *Can we strategically synthesize intelligent test cases which can drive the dynamic verification engines to explore correct schedule state space?*
3) *Can we synthesize realistic and deep bugs in concurrent programs to assess the quality of dynamic verification tools on a large number of subjects?*

II. Progress

We have attempted to expound the first two problem statements; the third problem is proposed in section V. We first present an overview of the technique designed for the deadlock detection (published [15]) problem and then the test case synthesis problem. Both approaches use constraint solving at heart.

A. HERMES*: Scalable Deadlock detection [15]*

We explore the first problem in the context of the Message Passing programming paradigm. Message Passing Interface (MPI) is the pioneer standard for distributed systems to communicate. *Communication deadlocks* in MPI programs is one of the prominent problems and it has been studied extensively in the past [28]. Due to the asynchronous nature and *wildcard* primitive calls in this paradigm, the prevalent nondeterministic communication of data can affect the control-flow of the program (*e.g.*, when the communicated data to a wildcard receive is used in a subsequent branch instruction of the program). Programs with the pattern mentioned above are termed as *multi-path* programs [32], and they significantly affect the scalability of existing verification techniques.

Our technique combines the strengths of dynamic verification and trace verification techniques, which provides scalability and multi-path coverage over these techniques, respectively.

Our hybrid method which discovers deadlocks in multi-path MPI programs exhaustively explores the executions of the program under a fixed input as follows: (i) it obtains a concrete run ρ of the program via dynamic analysis (via a scheduler that orchestrates a run); (ii) it encodes symbolically the set of *feasible runs* obtained from the same set of events as observed in ρ such that each process triggers the same control-flow decisions and executes the same sequence of communication calls as in ρ (note that the encoding captures the entire set of runtime matches of communication events from ρ); (iii) it checks for violations of any property (in our case, communication deadlocks); and (iv) if no property is violated, it alters the symbolic encoding to explore the feasibility of taking an alternate control flow path which is different from ρ. In case of such feasibility, it initiates a different concrete run [15].

Consider the program shown in Figure 1. It is a nondeterministic, multi-path, and deadlock-free program. The non-colored lines illustrate the pseudo-code of the program. The nondeterministic matching choice of $R1$ governs the execution of one of the multiple control flow branches.

Our approach statically discovers the code locations where the received data or message tag (a field in MPI send and receive calls that serve as a unique marker for messages) are used to branch at conditional statements. At these locations, we instrument certain calls to a *scheduler* (shown in blue color in Figure 1). The *scheduler* schedules the MPI calls of the program according to the MPI semantics and drives the execution. The scheduler is also responsible for building a partially ordered *happens-before* relation between these calls. At runtime, the instrumented code communicates the predicate expression in the branching instruction to the scheduler. Based on a trace ρ the symbolic encoding is generated from the execution of the program with instrumented code. This formula encodes all the semantically possible schedules of events observed in ρ, which

```
          Process 0                                     Process 1
Recv(*, x); //R1                               Send(P0, 10); //S1
if (x==10)                                              Process 2
  Recv(*, y); //R2                             Send(P0, 10); //S2
  toScheduler('x==10');                                 Process 3
else if (x==20)                                Send(P0, 30); //S3
  Recv(*, y); //R3
  toScheduler('x==20');
else if (x==30)
  Recv(*, y); //R4
  toScheduler('x==30');
Recv(*, z); //R5
```

Fig. 1: Instrumented *non single-path* program

follow the same control-flow decisions as made in ρ. A Satisfiability Modulo Theories (SMT) solver is used to solve this formula, which checks for the violation of the safety property.

In the example shown in Figure 1, `Process 0` executes the first control flow branch if $R1$ matches with either $S1$ or $S2$. If there is no property violation for this SMT formula, we verify another control flow path that may have been taken if $R1$ had matched with some other send.

To this effect, we want to change the path condition obtained from the trace to reschedule another execution through an unvisited control-flow. Hence, we alter the path condition (in a typical symbolic execution style) and execute the program again, so that it follows the path corresponding to the altered path condition. To force the scheduler to follow a different control-flow branch, we may also have to force a wildcard receive call to match with a send call that sends data different from the send call that matched before. We repeat this process until all the paths in the program are exhausted. In the context of the above example, in the second execution, $R1$ must match $S3$, and it must avoid matching $S2$ because $S2$ is sending the same data as $S1$ ($S1$ had already matched with $R1$ in the first execution).

The example program has six possible interleavings across multiple control flow paths. Our technique executes the program only twice to cover all of them. It thus shows the contrasting difference from trace verification, which does not provide full path coverage and from dynamic verification, which executes the program as many times as there are possible interleavings.

B. REVELIO*: Test Case Synthesis*

We seek solution to the test case synthesis problem in the context of multi-threaded Java programs. We present a novel approach to synthesize test cases for exposing communication deadlocks in these programs. A multi-threaded system is said to be caught in a communication deadlock when one of the threads is suspended because of a `wait` call, and none of the other threads can send a `notify` signal to wake it, so the system does not proceed [9], [12].

The technique to synthesize test cases for efficient schedule space navigation is implemented in a prototype tool called REVELIO. The approach has two key steps: (i) finding two distinct single-threaded execution traces and determining whether by combining these traces, a `wait-notify` communication deadlock can be exposed, and (ii) if the bug could indeed be manifested, then assembling these execution traces under specific input parameters to form a multi-threaded test case. Combining two separate sequential traces requires finding a common starting state from where two threads can spawn, eventually allowing the defect to manifest.

REVELIO's *Trace Extractor* component executes every non-private API of a class under test (CUT) with a symbolic execution engine [16] so that the paths leading to a `wait` or a `notify` call can be encountered. The explored traces along with the *path conditions* are stored in internal data structures.

From this set of traces, the *Match Predictor* component selects those pairs of traces that are compatible based on the data-types of their caller objects. A pair of traces with data-type compatible caller objects of `wait` and `notify` is eligible to be invoked from a single multi-threaded test case because they potentially comply with the communication protocol of the library. From the data-type compatible pairs of traces, we select those pairs that have the potential to expose a bug. The algorithm leverages an SMT solver to ascertain such pairs of traces. An encoding of the data-type compatible pair of traces (as a first-order logic formula) is shipped to a solver. The result of the solver determines whether there is a chance that the input traces can be combined into a test case.

If the solver returns SAT, then the set of methods is determined that must be invoked before the creation of threads such that a program state with the necessary environment is reached for manifesting the bug. We refer to these methods as *auxiliary methods*. Finding these auxiliary methods is the job of the component *Context Setter*. The last step is to create the test, which will invoke the found method call sequences from two threads. *Synthesizer* is responsible for generating such a test case. Post-factum validations of synthesized test are performed using the model checker JPF [35] in component *Verifier*.

III. RELATED WORK

Deadlock detection: Deadlock detection in message-passing programs is an active research domain with a rich body of literature. ISP [34] and DAMPI [36] are dynamic verifiers that enumerate all relevant executions of a program under a given input by rerunning the program. While ISP has a centralized scheduler, DAMPI has a distributed one.

Predictive trace analysis for multi-threaded C/C++ programs is another popular area of work. The central idea in these techniques is to encode the interleavings of program execution in a first-order logic formula [2], [37]. The work in [2] motivated the predictive trace analysis work for MPI, MCAPI, and CSP/CCS programs [5], [10], [11], [19], [32]. We base our encoding rules on the encoding presented by Forejt et al. [32], but their technique is restricted to *single-path* programs.

To improvise it for multi-path MPI programs, we executed the program multiple times with different path condition from the earlier runs as in Concolic Testing [3], [27]. However, a fair comparison of HERMES with concolic execution techniques cannot be performed since HERMES does not consider every conditional statement to be included in the path condition. HERMES uses data-aware analysis to prune irrelevant control-flow branches of the program.

CIVL [18] is an intermediate language to capture concurrency semantics of a set of concurrency dialects such as OpenMP, Pthreads, CUDA, and MPI. The back-end verifier can statically check properties such as functional correctness, deadlocks, and adherence to the rules of the MPI standard. CIVL creates a model of the program using symbolic execution and then uses model checking. HERMES creates a model of a single path of the program and uses symbolic encoding to verify that path of the program.

There has been much research in the area of deadlock detection of not just MPI programs, but multithreaded programs also [9], [12].

Test synthesis: A recent survey by Terragni et al. classified test synthesis techniques into three major categories: 1. Random-based, 2. Coverage-based, 3. Sequential-test based [31]. Whichever the category, the generated test cases can be divided into two parts. The part which is common to all the threads and is assembled before invoking the threads is called the *prefix* and the part which is a spawned thread that calls one of the public API methods is called the *suffix*. Random-based techniques generate test cases by randomly appending any suffixes to a prefix with random input parameters but without any specific intuition [20], [22]. These tools can generate as many tests as specified by the user. The random nature of the technique often generates, albeit fast, redundant tests, which, therefore, wastes verification resources.

Coverage-based techniques are goal-oriented and improve over random-based methods. These techniques aim to achieve high interleaving coverage for which they pre-compute the coverage requirements based on a coverage-driven heuristic [4], [30]. For example, the technique proposed by Lu et al. [4] tries to find and engage those method pairs in the test, which have been infrequently paired up before in the test-generation process. Their technique does not focus on a particular kind of concurrency bugs. The advantage of coverage-based techniques is that they restrict the generation of redundant tests. The disadvantage is that they rely on a heuristic, which might give false information (especially in case of statically computed) or might miss coverage requirements.

Sequential-test based techniques utilize sequential test cases provided with the library or the sequential tests generated with tools such as Randoop [21]. Samak et al. present a series of works to synthesize multi-threaded test cases for exposing atomicity violations [25], resource deadlocks [24], and races [26]. Sequential tests are not always adequate to exercise parallel constructs in the code. The `wait-notify` constructs are almost always guided by conditions expressed over the variables which have to be succinctly initialized. Moreover, each of their work is aimed at a specific type of a concurrency bug and none target communication deadlocks.

IV. RESULTS

To answer the first problem statement, we set the evaluation of HERMES [13] in the context of C/C++ MPI programs and compared it against the state-of-the-art verification tools (Mopper, Aislinn, ISP, and CIVL) to assess its efficiency and effectiveness. We used the FEVS test-suite [29] and benchmarks from prior research papers [1], [8], [33], [38]. On most single-path benchmarks, the performance of HERMES is comparable to Mopper and considerably better than the other state-of-the-art explicit-state model checkers without compromising on error-reporting. Benchmark `2D Diffusion` exhibits a complex communication pattern and a high degree of nondeterminism, which leads to a huge M^+. Hence, symbolic analysis tools did not perform well for such benchmarks. Evaluation with multi-path programs required us to compare HERMES with all tools except Mopper, since Mopper is constrained to work with only single-path programs. The basis for comparing against ISP was the number of times a program is executed. In contrast, the basis for comparing with other tools was the time taken to complete the verification. The results indicate that when the number of processes increases, the growth in execution time is relatively reasonable in HERMES in comparison with ISP and Aislinn. The scalability of HERMES regarding the number of processes comes from the fact that it prunes out the redundant paths and explores only the feasible ones.

For the second research question, we evaluated REVELIO [14] on twelve classes that are part of the real-world concurrent Java libraries. We gathered half of these subjects by exploring the issue repositories of these libraries and others from the literature [9], [12], [17]. These libraries profusely use `wait-notify` calls to communicate.

We compared our tool with general-purpose tools for Java programs that are available for the research community: a random-based test synthesizer, ConTeGe [22], and a coverage-based test synthesizer, CovCon [4].

We compared REVELIO with ConTeGe and CovCon based on their ability to synthesize a successful test (B) and the time they take to produce those tests. A successful test is the one that exposes a communication deadlock. REVELIO can synthesize successful tests for all twelve subjects as compared to ConTeGe, which was successful for one of them and CovCon, which was not successful in capturing any `wait-notify` error (under the chosen settings), but found thread-safety violations of different kinds for three of the subjects. REVELIO did not take more than 180 seconds for any of the subjects. In comparison, ConTeGe ran out of 600 seconds for four of the subjects. For the rest of the subjects, it terminated in less than this time frame and found no violations. For one of the issues, it crashed. Similar is the case with CovCon.

The number of compatible pairs of traces that exhibited a concurrency error was very less, considerably lesser than ConTeGe and CovCon, which generated thousands of tests. This precision can be attributed to the fact that our test synthesis approach is directed and goal-oriented. Although the encoding scheme depends on the over-approximate set of variable bindings, it does well in practice to search precisely for the candidate pairs of traces.

V. PROPOSED WORK

There has been an extensive effort in developing static, dynamic, and hybrid techniques to detect concurrency errors, namely races, order violations, atomicity violations, and deadlocks in concurrent programs [6]. Researchers test their bug detection or verification tools on a limited set of benchmarks. These benchmarks are generally taken from previous research papers, or bug repositories are explored to find bug reports that can be used for evaluating the appropriate use of the technique developed. Digging bug repositories is a time and effort consuming task, nonetheless inevitable. The survey by Fu et al. shows that most of the literature on concurrency bug avoidance, exposing, detection, and fixing focus on a total of just nine C/C++ libraries [6]. This indicates that more diversity in terms of bugs in the benchmarks is desired.

These benchmarks only provide information about the bugs that the detection tools can detect, but not the information about the bugs they miss. Moreover, the value of any individual dataset drops over time as tools adapt to it. Hence, there is a need to automatically synthesize bugs in the code to eradicate the problems mentioned above.

Roy et al. [23] proposed a technique to synthesize bugs in sequential programs. These bugs are injected deep inside the code by instrumenting an Error Transition System (ETS) that they find using a constraint-based approach. The idea is to instrument intelligent predicates in the code that can constrain the input, which can trigger the bug.

We take inspiration from this work to synthesize concurrency bugs into multithreaded programs. The main difference that separates multithreaded programs from sequential programs is the *nondeterminism*, which gives rise to multiple schedules for a single input, and a concurrency error can manifest in either of these interleavings. To synthesize a bug deep inside a concurrent multithreaded program, we not only have to tighten the input space at each state of the ETS, but also the interleaving space. This strategy will ensure that the bug detection tools are given an appropriate challenge and a fair chance to prove their worth.

The fault injection techniques such as Safire [7] focuses on injecting the instruction-based fault model in the system. They inject soft errors in the program by flipping a random bit in a random register. This bit-flipping introduces a small variation, and the error cascades to the instruction and the processor. This approach is unguided. In contrast to this, we are working towards systematically synthesizing intelligent, deep, and realistic bugs in the program. Specifically, we need to have the following necessary qualities in our bug synthesis technique:

1) the bug must manifest on a real input,
2) it should be rare, that means it should only manifest on some of the interleavings for a particular input,
3) it should be randomly distributed in the state space so that it does not favor a particular detection strategy.

ACKNOWLEDGMENT

The first author is grateful to her advisors Dr. Rahul Purandare and Dr. Subodh Sharma, for their valuable guidance. The thesis work of the first author is supported by the grant from

Tata Consultancy Services. The authors thank the anonymous reviewers for their valuable feedback for the published work.

References

[1] S. Böhm, O. Meca, and P. Jančar. *State-Space Reduction of Non-deterministically Synchronizing Systems Applicable to Deadlock Detection in MPI*, pages 102–118. 2016.

[2] Wang C., Kundu S., Ganai M K., and Gupta A. Symbolic predictive analysis for concurrent programs. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 256–272, 2009.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[4] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. ICSE '17, page 266–277, 2017.

[5] M. Elwakil, Z. Yang, and L. Wang. *CRI: Symbolic Debugger for MCAPI Applications*, pages 353–358. 2010.

[6] Haojie Fu, Zan Wang, Xiang Chen, and Xiangyu Fan. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Quality Journal*, 26(3):855–889, September 2018.

[7] G. Georgakoudis, I. Laguna, H. Vandierendonck, D. S. Nikolopoulos, and M. Schulz. Safire: Scalable and accurate fault injection for parallel multithreaded applications. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 890–899, 2019.

[8] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. 1999.

[9] David Hovemeyer and William Pugh. Finding concurrency bugs in java. In *In Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[10] Y. Huang and Mercer E. *Detecting MPI Zero Buffer Incompatibility by SMT Encoding*, pages 219–233. 2015.

[11] Y. Huang, Mercer E., and McCarthy J. Proving MCAPI executions are correct using SMT. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 26–36, 2013.

[12] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 327–336, 2010.

[13] Dhriti Khanna. Hermes. https://github.com/DhritiKhanna/Hermes, 2018.

[14] Dhriti Khanna. Revelio. https://github.com/DhritiKhanna/Revelio, 2020.

[15] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare. Dynamic symbolic verification of mpi programs. In *Formal Methods*, pages 466–484, Cham, 2018.

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

[17] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. Jacontebe: A benchmark suite of real-world java concurrency bugs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 178–189, 2015.

[18] Z. Luo, M. Zheng, and S F. Siegel. Verification of mpi programs using CIVL. In *Proceedings of the 24th European MPI Users' Group Meeting*, EuroMPI '17, pages 6:1–6:11, 2017.

[19] G Narayanaswamy. When truth is efficient: Analysing concurrency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 141–152, 2015.

[20] Adrian Nistor, Qingzhou Luo, Michael Pradel, Thomas R. Gross, and Darko Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. ICSE '12, page 727–737, 2012.

[21] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. OOPSLA '07, page 815–816, 2007.

[22] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. PLDI '12, page 521–530, 2012.

[23] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 224–234, 2018.

[24] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. OOPSLA '14, page 473–489, 2014.

[25] Malavika Samak and Murali Krishna Ramanathan. Synthesizing tests for detecting atomicity violations. ESEC/FSE 2015, page 131–142, 2015.

[26] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. PLDI '15, page 175–185, 2015.

[27] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, 2005.

[28] S V. Sharma, G. Gopalakrishnan, and R M. Kirby. A survey of MPI related debuggers and tools. 2007.

[29] S F. Siegel and Timothy K. Zirkel. Fevs: A functional equivalence verification suite for high-performance scientific computing. *Mathematics in Computer Science*, pages 427–435, Dec 2011.

[30] Valerio Terragni and Shing-Chi Cheung. Coverage-driven test code generation for concurrent classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1121–1132, 2016.

[31] Valerio Terragni and Mauro Pezzè. Effectiveness and challenges in generating concurrent tests for thread-safe classes. ASE 2018, page 64–75, 2018.

[32] Forejt V., Joshi S., Kroening D., Narayanaswamy G., and Sharma S. Precise predictive analysis for discovering communication deadlocks in MPI programs. *ACM Trans. Program. Lang. Syst.*, 39(4):15:1–15:27, 2017.

[33] S. Vakkalanka. *Efficient Dynamic Verification Algorithms for MPI Applications*. PhD thesis, 2010.

[34] S Vakkalanka, G Gopalakrishnan, and R M. Kirby. *Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings*, pages 66–79. 2008.

[35] Willem Visser, Corina S. Pundefinedsundefinedreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, page 97–107, 2004.

[36] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B R. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for mpi programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–10, 2010.

[37] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 328–342, 2010.

[38] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. Mpiwiz: Subgroup reproducible replay of mpi applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 251–260, 2009.