Combining Symbolic Execution and Model Checking to Verify MPI Programs

Hengbiao Yu
College of Computer, National University of Defense Technology, Changsha, China
State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
hengbiaoyu@nudt.edu.cn

ABSTRACT

Message Passing Interface (MPI) has become the standard programming paradigm in high performance computing. It is challenging to verify MPI programs because of high parallelism and nondeterminism. This paper presents MPI symbolic verifier (MPI-SV), the first symbolic execution based tool for verifying MPI programs having both blocking and non-blocking operations. MPI-SV exploits symbolic execution to automatically generate *path-level* models, and performs model checking on the models *w.r.t.* expected properties. The results of model checking are leveraged to prune redundant paths. We have implemented MPI-SV and the extensive evaluation demonstrates MPI-SV's effectiveness and efficiency.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**;

KEYWORDS

MPI; Verification; Symbolic Execution; Model Checking

ACM Reference Format:
Hengbiao Yu. 2018. Combining Symbolic Execution and Model Checking to Verify MPI Programs. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3183440.3190336

1 INTRODUCTION

The Message Passing Interface (MPI) [21] has been widely adopted for developing high performance computing applications. Complex program features, such as *non-determinism* and *non-blocking*, make MPI programs error-prone. How to improve the reliability of MPI programs has been a challenging problem [11].

Static approaches [1, 2, 14, 19] perform analysis on soundly abstracted models, and suffer from false positives, manual effort, or complex MPI features. In addition, Dynamic approaches [18, 23] run the program concretely and exploit the runtime information for analysis. As a result, dynamic methods are confined by the poor input coverage. Compared with static and dynamic approaches, symbolic execution [9, 13] achieves better abstraction precision and input coverage, respectively. TASS [20] and MPISE [8] are existing symbolic execution based approaches, but they do not support non-blocking MPI programs, which are ubiquitous in real world MPI programs.

This paper presents MPI-SV, the first symbolic execution based tool for verifying MPI programs having both non-blocking and non-deterministic MPI operations. To improve the scalability, MPI-SV integrates symbolic execution with model checking [5]. More precisely, symbolic execution systematically abstracts control and data dependencies to generate verifiable *path-level* models for model checking, and the results of model checking are leveraged to prune redundant paths for symbolic execution.

We have implemented MPI-SV for MPI C programs based on Cloud9 [3] and PAT [22], and evaluated it on 11 real world programs *w.r.t.* deadlock freedom property. The evaluation results are promising and demonstrate MPI-SV's effectiveness and efficiency.

2 MPI SYMBOLIC VERIFICATION

An MPI program runs in many processes, which communicate through message passing in either blocking or non-blocking manner. First, we introduce three key MPI operations: `Send(i)` sends a message to the *i*th process, and the process blocks until the message is copied into the local sending buffer[1]; `Recv(i)` receives a message from the *i*th process, and the process blocks until the message from the *i*th process is received; and `IRecv(*,req)` receives a message from *any* process, and the process returns immediately after the operation is issued, where `req` is the handler of the operation. Non-blocking operations can cause out-of-order completion, and wildcard receives can cause non-determinism.

P_0	P_1	P_2	P_3

`Send(1)`	**if** (x != 'a')
	`Recv(0)`
	else
	`IRecv(*,req);`
	`Recv(3)`
		`Send(1)`	`Send(1)`

Figure 1: A motivating example of MPI programs.

The motivation example is run under 4 processes. Process P_0, P_2 and P_3 simply send a message to P_1 and terminate. If input x is not a, P_1 receives a message from P_0 in a blocking manner, otherwise receiving a message from any process in a non-blocking manner. Then P_1 uses a blocking receive to receive a message from P_3. Obviously, if x equals a and `IRecv(*,req)` matches the

[1]For brevity, we assume that the local buffer is infinite, hence the operation returns immediately after being issued.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05…$15.00
https://doi.org/10.1145/3183440.3190336

`Send(1)` in P_3, a deadlock happens because the `Recv(3)` in P_1 has no matching.[2]

MPI Symbolic Execution. Considering MPI processes are memory independent, MPI-SV will select a process to execute in a round-robin manner to avoid exploring full interleavings. Specifically, a process keeps running until blocks or terminates, and the encountered MPI operations are recorded instead of executed. Once all processes block or terminate, MPI-SV handles the possible message passings *w.r.t.* the MPI standard [7]. To ensure soundness, in addition to branches, symbolic execution forks states for different message matchings and essential process interleavings, *e.g.*, in the false branch, since `IRecv(*,req)` can match the send operation from P_0, P_2 or P_3, three states will be forked. Hence, symbolic execution detects the deadlock in the $4th$ iteration.

CSP Modeling based Boosting. Once a *violation-free* path p is generated, MPI-SV automatically builds a Communication Sequential Process (CSP) [17] model $\mathcal{M}$ that represents the equivalent paths of p by changing the interleavings and matchings of the communication operations in p. The key idea of modeling is : (1) using channel reading and writing to model the message send and receive; (2) using external choice to model the non-determinism of wildcard receive; and (3) using parallel composition to model the parallelism. We have proved that the CSP modeling method is sound and complete. Given a property φ, if $\mathcal{M} \models \varphi$, MPI-SV can prune the states forked by different matchings and interleavings along p due to the soundness; otherwise, if model checker gives a counter-example, MPI-SV reports a violation due to the completeness.

Violation detection. After exploring the first path in the false branch, where `IRecv(*,req)` matches the `Send(1)` in P_0, MPI-SV builds a model $\mathcal{M}$, and feeds it to the CSP model checker PAT to verify deadlock freedom. Obviously, PAT reports a counter-example where `IRecv(*,req)` matches the `Send(1)` in P_3. Hence, MPI-SV only needs 2 iterations to detect the deadlock.

Pruning. Suppose we replace the `Recv(3)` with `Recv(*)`, the program becomes deadlock free. Pure symbolic execution needs 8 iterations to complete verification, *i.e.*, 2 paths for the true branch and 6 paths for the false branch. While MPI-SV builds a CSP model after exploring the first path in either the true or false branch. The model is verified deadlock free by PAT, and MPI-SV prunes the states forked by different matchings of wildcard receives. Hence, MPI-SV only needs to explore 2 paths to verify deadlock freedom.

Properties. Symbolic execution using partial order reduction (POR) [10] is sound *w.r.t.* global reachability properties, *e.g.*, deadlock freedom. However, with the help of CSP modeling and model checking, MPI-SV can soundly verify any model checker supported properties of communication behaviors, except existential path quantifier properties [4], which can be supported by adjusting MPI-SV's framework.

3 EVALUATION

To evaluate MPI-SV, we apply it to verify 11 real-world open source MPI C programs (shown in Table 1) and their mutants[3] *w.r.t.* deadlock freedom. The time threshold of verification is 1 hour, and we run each task under two configurations: pure symbolic execution using DFS and MPI-SV.

Table 1: The programs in the experiments

Program	LOC	Brief Description
`DTG`	90	Dependence transition group
`Integrate_mw`	181	Integral computing
`Diffusion2d`	197	Simulation of diffusion equation
`Gauss_elim`	341	Gaussian elimination
`Heat`	613	Heat equation solver
`Pingpong`	220	Comm performance testing
`Mandelbrot`	268	Mandelbrot set drawing
`Image_manip`	360	Image manipulation
`DepSolver`	8988	Electrostatic solver
`Kfray`	12728	KF-Ray parallel raytracer
`ClustalW`	23265	Multiple sequence alignment
Total	**47251**	**11 open source programs**

For the 108 tasks, MPI-SV can complete 94 tasks (87%), where completion means successfully verifying a program or finding a deadlock. While pure symbolic execution completes 61 tasks (56%), with a 31% improvement. Specifically, compared with pure symbolic execution, MPI-SV achieves an average 7.25X speedup for completing a program's path space, and an average 2.64X speedup for detecting deadlocks.

4 RELATED WORK AND CONCLUSION

MOPPER [6] encodes the deadlock detection problem under a given input in a SAT equation. Similarly, Huang *et al.* propose an SMT encoding method for deadlock detection [12]. Compared with them, MPI-SV adopts CSP constructs to simulate a path, which enables a more expressive modeling, *e.g.*, supporting the conditional completes-before [23]. Besides, MPI-SV can verify safety and liveness properties in LTL [16]. CIVL [15] uses symbolic execution to build a model for the whole MPI program, and performs model checking on the model. While MPI-SV uses symbolic execution to extract *path-level* models, which can ease the burden of model checking. In addition, CIVL does not support non-blocking MPI operations.

We have presented MPI-SV, a framework that integrates symbolic execution with model checking for verifying real world MPI programs having non-blocking and non-deterministic operations. We have implemented MPI-SV, and the experimental results demonstrate MPI-SV's effectiveness and efficiency. Future work lies in two directions: (1) improving MPI-SV to handle more complicated MPI programs, *e.g.*, the programs using dynamic scheduling; and (2) enhancing the feasibility, usability and robustness of MPI-SV, and releasing it to benefit the community.

ACKNOWLEDGEMENT

This research was supported by NSFC Program (No. 61472440, 61632015, 61690203).

[2] According to the MPI standard, if two receive operations can match the same message, the message should match the first issued one.

[3] We randomly replace a deterministic receive with a wildcard receive, and vice versa. We generate five mutants for a program if possible.

REFERENCES

[1] Vincent Botbol, Emmanuel Chailloux, and Tristan Le Gall. 2017. Static Analysis of Communicating Processes Using Symbolic Transducers. In *VMCAI*. 73–90.

[2] Greg Bronevetsky. 2009. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*. 1–12.

[3] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. 2010. Cloud9: a software testing service. *ACM SIGOPS Operating Systems Review* 43, 4 (2010), 5–10.

[4] Edmund M. Clarke and E. Allen Emerson. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs*. 52–71.

[5] Edmund M Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT press.

[6] Vojtěch Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2014. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *FM*. 263–278.

[7] MPI Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. http://mpi-forum.org. (2012).

[8] Xianjin Fu, Zhenbang Chen, Yufeng Zhang, Chun Huang, Wei Dong, and Ji Wang. 2015. MPISE: Symbolic Execution of MPI Programs. In *HASE*. 181–188.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. 213–223.

[10] Patrice Godefroid, J Van Leeuwen, J Hartmanis, G Goos, and Pierre Wolper. 1996. Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. *Lecture Notes in Computer Science* (1996).

[11] Ganesh Gopalakrishnan, Robert M. Kirby, Stephen F. Siegel, Rajeev Thakur, William Gropp, Ewing L. Lusk, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2011. Formal analysis of MPI-based parallel programs. *Commun. ACM* (2011), 82–91.

[12] Yu Huang and Eric Mercer. 2015. Detecting MPI Zero Buffer Incompatibility by SMT Encoding. In *NFM*. 219–233.

[13] J.C. King. 1976. Symbolic execution and program testing. *Commun. ACM* (1976), 385–394.

[14] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *OOPSLA*. 280–298.

[15] Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. 2017. Verification of MPI programs using CIVL. In *EuroMPI*. 6:1–6:11.

[16] Zohar Manna and Amir Pnueli. 1992. *The temporal logic of reactive and concurrent systems - specification*. Springer.

[17] Bill Roscoe. 2005. *The theory and practice of concurrency*. Prentice-Hall.

[18] Victor Samofalov, V. Krukov, B. Kuhn, S. Zheltov, Alexander V. Konovalov, and J. DeSouza. 2005. Automated Correctness Analysis of MPI Programs with Intel(r) Message Checker. In *PARCO*. 901–908.

[19] Stephen F. Siegel. 2007. Verifying Parallel Programs with MPI-Spin. In *PVM/MPI*. 13–14.

[20] Stephen F. Siegel and Timothy K. Zirkel. 2011. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science* (2011), 395–426.

[21] Marc Snir. 1998. *MPI–the Complete Reference: The MPI core*. Vol. 1. MIT press.

[22] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *CAV*. 709–714.

[23] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Dynamic Verification of MPI Programs with Reductions in Presence of Split Operations and Relaxed Orderings. In *CAV*. 66–79.