MPI-SV: A Symbolic Verifier for MPI Programs

Zhenbang Chen[1], Hengbiao Yu[1], Xianjin Fu[1,2], Ji Wang[1,2]
[1]College of Computer, National University of Defense Technology, Changsha, China
[2]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
{zbchen,hengbiaoyu,wj}@nudt.edu.cn

ABSTRACT

Message passing is the primary programming paradigm in high-performance computing. However, developing message passing programs is challenging due to the non-determinism caused by parallel execution and complex programming features such as non-deterministic communications and asynchrony. We present MPI-SV, a symbolic verifier for verifying the parallel C programs using message passing interface (MPI). MPI-SV combines symbolic execution and model checking in a synergistic manner to improve the scalability and enlarge the scope of verifiable properties. We have applied MPI-SV to real-world MPI C programs. The experimental results indicate that MPI-SV can, on average, achieve 19x speedups in verifying deadlock-freedom and 5x speedups in finding counter-examples. MPI-SV can be accessed at https://mpi-sv.github.io, and the demonstration video is at https://youtu.be/zzCY0CPDNCw.

CCS CONCEPTS

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

KEYWORDS

Symbolic Verification; Message Passing Interface; MPI-SV

ACM Reference Format:
Zhenbang Chen, Hengbiao Yu, Xianjin Fu, Ji Wang. 2020. MPI-SV: A Symbolic Verifier for MPI Programs. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3377812.3382144

1 INTRODUCTION

Message Passing Interface (MPI) [5] is the current *de facto* standard for developing the parallel applications in high-performance computing. Developing MPI applications is challenging [7] due to the nature of MPI programming's complexities, such as non-determinism and non-blocking communications. Ensuring the correctness of MPI programs is highly demanded [7].

Existing verification tools for MPI programs are mainly dynamic ones [18][4]. These dynamic verification tools analyze the correctness of the MPI program under a specific input; hence, they may miss the bugs depending on program inputs. On the other hand, although there exist static verification tools (such as MPI-SPIN [15], TASS [16] and CIVL [12]), these tools either need manual modeling or do not support the MPI programs with non-blocking operations, which are ubiquitous in real-world MPI programs [8]. Besides, existing dynamic or static automatic verifiers of MPI programs can only support the verification of reachability properties.

We present in this paper MPI-SV, *i.e.*, a symbolic verifier for MPI C programs. MPI-SV covers the non-determinism caused by program inputs and supports the verification of the MPI programs with non-blocking and non-deterministic operations; besides, MPI-SV can verify LTL properties [3]. Insides MPI-SV, we have implemented our technique in [19], which combines symbolic execution and model checking in a synergistic manner to enlarge the scope of verifiable properties and improve the scalability of verification.

We have evaluated MPI-SV on real-world open-source MPI programs. The experimental results indicate that MPI-SV is effective and efficient for verifying MPI programs. Especially, insides the benchmark, there are *three* large programs that are beyond the ability of any existing *static automatic* verifiers for MPI programs.

2 BACKGROUND AND OVERVIEW

This section briefly introduces the basic concepts of MPI programs and the verification technique in MPI-SV.

2.1 MPI Programs

An MPI program can be coded in different languages, such as C, C++, and FORTRAN. We run the program in a fixed number of *processes* that can be spanned on one or multiple machines. These processes are running in parallel and coordinate by messages passing to accomplish a computation task. There are following two kinds *atomic* MPI APIs for message passing.

- **Blocking operations**, including `Barrier`, `MPI_Ssend`, `MPI_Send`, `MPI_Recv`, `MPI_Wait`, *etc.* Invoking any of these operations will block the process until the operation is finished. For example, `MPI_Ssend` sends a message to a destination process and blocks the process until the message is well-received by the receiver.
- **Non-blocking operations**, *e.g.*, `MPI_Isend` and `MPI_Irecv`. The execution of a non-blocking operation does not block the process. The status of the operation will be checked later (*e.g.*, using a `Wait` operation) before using the operation's message. Non-blocking operations are commonly used to improve the MPI application's performance.

The non-deterministic communication operations are wildcard receives, *i.e.*, `MPI_Recv` and `MPI_Irecv` with any source indicator (represented by `MPI_ANY_SOURCE`). A wildcard receive operation o will receive any message that other processes send to o's process; hence, when multiple processes send a message to o's process, o will only receive one, which results in non-determinism. Figure 1

ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382144

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv) {
  int rank = -1;
  MPI_Init(&argc, &argv); //init
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); // get rank
  if (rank == 0) {
    char c;
    scanf("%c", &c);
    int v1, v2;
    if (c != 'a') {
        MPI_Recv(&v1, ..., 1, ...);
    } else {
        MPI_IRecv(&v1, ..., MPI_ANY_SOURCE, ...);
    }
    MPI_Recv(&v2, ..., 3, ...);
  } else {
    MPI_Send (&rank, ..., 0, ...);
  }
  MPI_Finalize ();
  return 0;
}
```

Figure 1: An example MPI C program

shows an MPI program. For the sake of the space limit, we omit some parts that are not related to demonstration.

The first process (*i.e.*, the one with `rank` 0, denoted by P_0) gets an input character `c` first. Then, P_0 will receive `v1`'s value from P_1 if `c` is *not* equal to 'a'; otherwise, P_0 will use a non-blocking wildcard receive to receive the value. Finally, P_0 receives `v2`'s value from P_3. For the other processes, each sends its rank value to P_0 and terminates. If we run this MPI program in four process, an error will happen if `c` is equal to 'a' and the `MPI_Irecv` receives the message from P_3. Then, P_0's last blocking receive blocks P_0 because the message from P_3 is already received, which results in a deadlock, *i.e.*, the program does not terminate but cannot progress.

As indicated by the example program, it is necessary to handle the non-determinism caused by both program input and wildcard receives to verify MPI programs. Besides, non-blocking operations also improve the complexity of handling wildcard receives.

2.2 Overview of MPI-SV

Figure 2 shows MPI-SV's high-level framework. The inputs contain an MPI program, the property φ to verify, and the number of processes. Inside MPI-SV, symbolic execution and model checking are combined in a synergy manner to verify MPI programs. MPI-SV uses symbolic execution to reason the control and data flows of the MPI program. When getting a *normally terminated* program path p, MPI-SV generates a CSP [14] model $\mathcal{M}$ representing p's equivalent program behavior. Then, MPI-SV uses a CSP model checker to verify the path model $\mathcal{M}$. If $\mathcal{M}$ does not satisfy φ, a counter-example is found; if $\mathcal{M}$ satisfies φ, MPI-SV continues symbolic execution.

Blocking-driven symbolic execution The challenge of the symbolic execution in MPI-SV is two folds: a) collecting the possibly matched send operations of wildcard operations; b) path explosion caused by the parallel execution. In [19], we propose blocking-driven symbolic execution (BDSE) for MPI programs. The key idea is to postpone the time for calculating the matching information *as later as possible* and employ partial order reduction (POR) [3] to avoid the full exploration of parallel executions. We have proven that BDSE is correct for reachability properties. However, even with POR, BDSE still suffers from path explosion problem because of symbolic execution's nature and is not correct for *non-reachability* properties, which is why MPI-SV needs CSP-based path modeling.

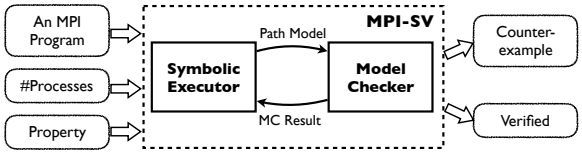

Figure 2: Framework of MPI-SV

CSP-based path modeling The modeling only models the communication behavior along a normally terminated path p. The modeling uses the channel operators in CSP to model message passing and the parallel operator to model non-blocking operations and parallel executions. The generated model encodes the equivalent communication behavior along p, *i.e.*, the behavior by only changing the matches of wildcard operations. We have proven that the modeling is precise *w.r.t.* the MPI standard. Based on this result, if the path model $\mathcal{M}$ does not satisfy the property φ, a true bug is found; if $\mathcal{M}$ satisfies φ, MPI-SV prunes p's the equivalent paths in symbolic execution, which improves the scalability of symbolic execution. For example, for the program in Figure 1, MPI-SV needs 2 paths to find the deadlock; however, MPI-SV without CSP modeling or model checking needs 4 paths to detect the deadlock. Besides improving scalability, CSP modeling also enlarges the scope of the verifiable properties, because the model represents all the possible interleavings of the communication operations along p. In principle, MPI-SV supports the properties that are supported by the underlying model checker. Now, MPI-SV uses CSP model checker PAT that supports LTL properties.

In summary, MPI-SV synergistically integrates symbolic execution and model checking to verify MPI programs. Both symbolic execution and model checking complement to each other. Symbolic execution handles the complex features of the program to generate verifiable models for model checking; meanwhile, model checking improves the scalability and enlarges the scope of verifiable properties for pure symbolic execution.

3 TOOL DESIGN AND IMPLEMENTATION

Figure 3 shows MPI-SV's architecture. To support the real-world MPI C programs, we built MPI-SV on Cloud9 [1] that is based on KLEE [2] and enhances KLEE with a better POSIX environment support and parallel symbolic execution. The input MPI C program needs be compiled into LLVM intermediate representation (IR) by Clang. Symbolic execution is carried out on the program's IR.

3.1 Executor

The executor is the core component in MPI-SV. The executor implements blocking-driven symbolic execution. Same as the traditional symbolic execution [11], the executor symbolically executes the IR program in a state-forking style. When encountering a branch statement with a symbolic condition, the executor forks the program state into two if both branches are feasible.

The executor analyzes the multi-threaded version of the MPI program, which is also the reason for using Cloud9 that supports

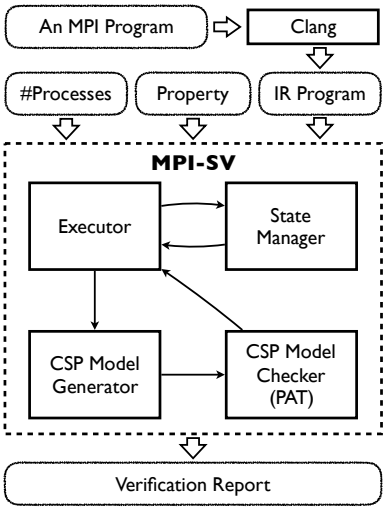

Figure 3: Architecture of MPI-SV

the analysis of multi-threaded C programs. Each MPI process is running as a thread in symbolic execution. Each program state (global state) is composed by the states of the processes. In principle, any process's advance changes the global state. To implement blocking-driven symbolic execution, the executor does not execute the encountered communications operations but records them and the calling stacks. When the global state blocks, *i.e.*, each process blocks, executor decides the matchings of the recorded MPI operations and executes the matched operations. In this way, the executor tries to get all the possible matchings of wildcard operations. Besides, leveraged by POR, executor always starts the execution of the non-blocked process with the *minimum* rank and avoids interleaving the statement executions of different processes.

Executor uses a multi-threaded MPI simulation library, *i.e.*, AzequiaMPI [13], as the environment support for MPI APIs. AzequiaMPI models a wide range of MPI APIs and supports the simulation of real-world MPI programs. Besides, global variables result in a problem for simulating the MPI program in the multi-threaded version because each process should have its own memory space for global variables. We solve this problem by maintaining each thread's own memory space of global variables.

3.2 State Manager

The state manager stores the explored states and controls the way of exploring the global state space of the MPI program. We inherit the search heuristics from KLEE, including DFS, BFS, Random+CoverNew, *etc.* The default strategy is DFS because using DFS is more possible to generate a terminated path, on which MPI-SV can use CSP modeling and model checking to boost the state exploration. When a path model is verified, the state manager is also in charge of pruning redundant states.

3.3 CSP model generator

The CSP model generator is in charge of generating CSP models for terminated paths. Given a global state in which all the processes normally terminate, the generator iterates the MPI operation sequence of each process (recorded during symbolic execution) to generate a corresponding CSP process. Then, the CSP processes are composed using the CSP parallel composition operator to form the global CSP model, which will be verified by the model checker. The generation has a polynomial-time complexity *w.r.t.* the total length of operation sequences. Please refer to [19] for the details.

3.4 CSP model checker

MPI-SV uses PAT [17] as the underlying CSP model checker. PAT supports the verification of the safety and liveness properties in LTL. In the implementation, the generator dumps a path model into a file that satisfies the input format of PAT. The executor invokes PAT to verify the model and parses the output for reporting counter-example or state pruning.

4 USAGE AND EVALUATION

To use MPI-SV, a user needs to compile the MPI program into IR first by using our Clang wrapped compiler script `mpisvcc`. For example,

```
mpisvcc demo.c -o demo.bc
```

Then, the user can use MPI-SV to analyze the IR program.

```
mpisv <#Procs> <Arg>* demo.bc <pArg>*
```

`<#Procs>` is the number of processes. `<Arg>*` is the list of MPI-SV's arguments. We use `-wild-opt` as the MPI-SV's argument to indicate using CSP modeling and model checking. `<pArgs>*` is the list of the analyzed program's arguments. The user can specify the verification property in LTL and pass the property file to MPI-SV. Deadlock freedom is the default property. To symbolize a variable, the user can use the symbolization function inherited from KLEE. For example, we can symbolize the variable `c` in Figure 1 as follows.

```
klee_make_symbolic(&c, sizeof(c), ``c'');
```

Readers can refer to MPI-SV's website for more details of usage.

We have evaluated MPI-SV on the 12 real-world open source MPI C programs in Table 1. The verified properties are deadlock-freedom and two application-dependent temporal properties.

Table 1: The programs in the experiments.

Program	LOC	Brief Description
DTG	90	Dependence transition group
Matmat	105	Matrix multiplication
Integrate	181	Integral computing
Diffusion2d	197	Simulation of diffusion equation
Gauss_elim	341	Gaussian elimination
Heat	613	Heat equation solver
Mandelbrot	268	Mandelbrot set drawing
Sorting	218	Array sorting
Image_manip	360	Image manipulation
DepSolver	8988	Multimaterial electrostatic solver
Kfray	12728	KF-Ray parallel raytracer
ClustalW	23265	Multiple sequence alignment
Total	**47354**	**12 open source programs**

Each verification task contains an MPI program, a fixed number of the processes, and the verification property. We set the verification time of each task to be one hour. In total, we have 111 deadlock

freedom verification tasks. Since there are no existing automatic static verification tools that support the MPI program with non-blocking operations, the primary baseline (denoted by `Baseline`) is the version of MPI-SV with pure symbolic execution, *i.e.*, without CSP modeling or model checking.

4.1 Effectiveness and efficiency

MPI-SV can complete 100 (90%) tasks, *i.e.*, finding a counter-example or proving that the MPI program satisfies the property under the fixed number of processes. `Baseline`, *i.e.*, pure symbolic execution, can complete 61 tasks. For the completed tasks in which a counter-example is found, MPI-SV achieves a 5x speedups on average; for the ones where verification succeeds, MPI-SV has a 19x speedups. These results indicate that the synergy between symbolic execution and model checking makes MPI-SV be both effective and efficient.

There are several tasks on which MPI-SV does not outperform `Baseline`. The reasons are: 1) the paths in these tasks contain a large number of wildcard operations, which makes the path model too complex to be verified by PAT; or 2) the paths have very few wildcard operations but the program has a huge path space, and the model checking cannot result in path pruning but brings overload.

4.2 Bug finding

MPI-SV is a tool that can help the developers of MPI applications to find bugs at the development stage. We use MPI-SV to find unknown runtime bugs (most are memory access out-of-bound bugs) in the benchmark programs in Table 1. We explain two here.

- **Bug 1**: Memory out-of-bound at `Line 213 in depSolver's gmre-mpi.c`. The code and the context is

```
for(i = 0; i < MAX_ITERS; i++){
  ...
  krylov[i+1] = doubleVector(size,1);
```

  `krylov` is an array whose size is `MAX_ITERS`; hence, the memory out-of-bound error happens when `i` is `MAT_ITERS - 1`.
- **Bug 2**: Memory out-of-bound at `Line 3095 in ClustalW's interface.c`. The code is

```
if (amino_acid_codes[seq_array[j][i+fres-1]]==c)
```

  `amino_acid_codes` is an array whose size is 26. The error will happen when `seq_array[j][i+fres-1]` is less than 0 or greater than 25.

MPI-SV does not find unknown deadlock bugs in the benchmark programs. Deadlock bugs are very severe and have usually be fixed during the development, especially for the benchmark programs that have been developed and used for a long time.

4.3 Other usage

MPI-SV has been used in [10] for creating a benchmark to evaluate existing model checkers. In total, the benchmark consists of 2318 path models automatically generated by MPI-SV from 10 MPI programs. Then, the benchmark is used to evaluate the representative model checkers, including SPIN, PAT, FDR, *etc.*

5 RELATED WORK

MPI-SV is related to the existing verification tools for MPI programs. Dynamic verification tools, such as ISP [6] and MOPPER [4], to name a few, run the MPI program under a specific input and verify the path to ensure the correctness under different matchings of wildcard operations. Compared with these tools, MPI-SV can cover the input space and supports a larger scope of verifiable properties.

Static verification tools abstract a model from the MPI program and verify the model. MPI-SPIN [15] needs to abstract the model manually; then, MPI-SPIN employs SPIN [9] to do the verification. CIVL [12] also uses symbolic execution to analyze MPI programs. CIVL translates the MPI program into its intermediate representation (IR) and does the symbolic execution of the IR program. CIVL does not support the MPI program with non-blocking operations and cannot verify non-reachability properties. Compared with MPI-SPIN and CIVL, MPI-SV is an automatic verifier that supports non-blocking MPI programs and LTL property verification.

ACKNOWLEDGEMENT

This research was supported by National Key R&D Program of China (No. 2017YFB1001802) and NSFC Program (No. 61902409, 61632015, 61690203, and 61532007).

REFERENCES

[1] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *EuroSYS*. 183–198.
[2] C. Cadar, D. Dunbar, and D. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*. 209–224.
[3] Edmund M Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT press.
[4] Vojtěch Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2014. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *FM*. 263–278.
[5] MPI Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. http://mpi-forum.org. (2012).
[6] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *CAV (Lecture Notes in Computer Science)*, Vol. 4590. Springer, 519–531.
[7] Ganesh Gopalakrishnan, Paul D. Hovland, Costin Iancu, Sriram Krishnamoorthy, Ignacio Laguna, Richard A. Lethin, Koushik Sen, Stephen F. Siegel, and Armando Solar-Lezama. 2017. Report of the HPC Correctness Summit Jan 25-26, 2017, Washington, DC. (2017).
[8] William Gropp, Ewing Lusk, and Rajeev Thakur. 1999. *Using MPI-2: Advanced features of the message-passing interface*. MIT press.
[9] Gerard J Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* (1997), 279–295.
[10] Weijiang Hong, Zhenbang Chen, Hengbiao Yu, and Ji Wang. 2019. Evaluation of model checkers by verifying message passing programs. *SCIENCE CHINA Information Sciences* 62, 10 (2019), 200101:1–200101:24.
[11] J.C. King. 1976. Symbolic execution and program testing. *Commun. ACM* (1976), 385–394.
[12] Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. 2017. Verification of MPI programs using CIVL. In *EuroMPI*. 6:1–6:11.
[13] Juan A. Rico-Gallego and Juan Carlos Díaz Martín. 2011. Performance Evaluation of Thread-Based MPI in Shared Memory. In *EuroMPI*. 337–338.
[14] Bill Roscoe. 2005. *The theory and practice of concurrency*. Prentice-Hall.
[15] Stephen F. Siegel. 2007. Model Checking Nonblocking MPI Programs. In *VMCAI*. 44–58.
[16] Stephen F. Siegel and Timothy K. Zirkel. 2011. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science* (2011), 395–426.
[17] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *CAV*. 709–714.
[18] 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.
[19] Hengbiao Yu, Zhenbang Chen, Xianjin Fu, Ji Wang, Zhendong Su, Jun Sun, Chun Huang, and Wei Dong. 2020. Symbolic Verification of Message Passing Interface Programs. In *ICSE*. to appear.