

# Symbolic Verification of Message Passing Interface Programs

Hengbiao Yu<sup>1\*</sup>, Zhenbang Chen<sup>1\*</sup>, Xianjin Fu<sup>1,2</sup>, Ji Wang<sup>1,2\*</sup>, Zhendong Su<sup>3</sup>,  
Jun Sun<sup>4</sup>, Chun Huang<sup>1</sup>, Wei Dong<sup>1</sup>

<sup>1</sup>College of Computer, National University of Defense Technology, Changsha, China

<sup>2</sup>State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China

<sup>3</sup>Department of Computer Science, ETH Zurich, Switzerland

<sup>4</sup>School of Information Systems, Singapore Management University, Singapore

{hengbiaoyu,zbchen,wj}@nudt.edu.cn,zhendong.su@inf.ethz.ch,junsun@smu.edu.sg,wdong@nudt.edu.cn

## ABSTRACT

Message passing is the standard paradigm of programming in high-performance computing. However, verifying Message Passing Interface (MPI) programs is challenging, due to the complex program features (such as non-determinism and non-blocking operations). In this work, we present MPI symbolic verifier (MPI-SV), the first symbolic execution based tool for automatically verifying MPI programs with non-blocking operations. MPI-SV combines symbolic execution and model checking in a synergistic way to tackle the challenges in MPI program verification. The synergy improves the scalability and enlarges the scope of verifiable properties. We have implemented MPI-SV<sup>1</sup> and evaluated it with 111 real-world MPI verification tasks. The pure symbolic execution-based technique successfully verifies 61 out of the 111 tasks (55%) within one hour, while in comparison, MPI-SV verifies 100 tasks (90%). On average, compared with pure symbolic execution, MPI-SV achieves 19x speedups on verifying the satisfaction of the critical property and 5x speedups on finding violations.

## CCS CONCEPTS

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

## KEYWORDS

Symbolic Verification; Symbolic Execution; Model Checking; Message Passing Interface; Synergy

## ACM Reference Format:

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 '20: ICSE '20: 42nd International Conference on Software Engineering*, May 23-29, 2020, Seoul, South Korea. ACM, New York, NY, USA, 13 pages. <https://doi.org/XX.XXXX/XXXXXXXX.XXXXXXX>

\*The first two authors contributed equally to this work and are co-first authors. Zhenbang Chen and Ji Wang are the corresponding authors.

<sup>1</sup>MPI-SV is available <https://mpi-sv.github.io>.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

ICSE '20, May 23-29, 2020, Seoul, South Korea

© 2020 Association for Computing Machinery.

ACM ISBN xxx-x-xxxx-xxxx-x/xx/xx...\$15.00

<https://doi.org/XX.XXXX/XXXXXXXX.XXXXXXX>

## 1 INTRODUCTION

Nowadays, an increasing number of high-performance computing (HPC) applications have been developed to solve large-scale problems [11]. The Message Passing Interface (MPI) [78] is the current *de facto* standard programming paradigm for developing HPC applications. Many MPI programs are developed with significant human effort. One of the reasons is that MPI programs are *error-prone* because of complex program features (such as *non-determinism* and *asynchrony*) and their scale. Improving the reliability of MPI programs is challenging [29, 30].

Program analysis [64] is an effective technique for improving program reliability. Existing methods for analyzing MPI programs can be categorized into *dynamic* and *static* approaches. Most existing methods are dynamic, such as debugging [51], correctness checking [71] and dynamic verification [83]. These methods need concrete inputs to run MPI programs and perform analysis based on runtime information. Hence, dynamic approaches may miss input-related program errors. Static approaches [5, 9, 55, 74] analyze abstract models of MPI programs and suffer from false alarms, manual effort, and poor scalability. To the best of our knowledge, existing *automated verification* approaches for MPI programs either do not support *input-related* analysis or fail to support the analysis of the MPI programs with *non-blocking* operations, the invocations of which do not block the program execution. Non-blocking operations are ubiquitous in real-world MPI programs for improving the performance but introduce more complexity to programming.

Symbolic execution [27, 48] supports input-related analysis by systematically exploring a program's path space. In principle, symbolic execution provides a balance between concrete execution and static abstraction with improved input coverage or more precise program abstraction. However, symbolic execution based analyses suffer from path explosion due to the exponential increase of program paths *w.r.t.* the number of conditional statements. The problem is particularly severe when analyzing MPI programs because of parallel execution and non-deterministic operations. Existing symbolic execution based verification approaches [77][25] do not support non-blocking MPI operations.

In this work, we present MPI-SV, a novel verifier for MPI programs by smartly integrating symbolic execution and model checking. As far as we know, MPI-SV is the first automated verifier that supports non-blocking MPI programs and LTL [58] property verification. MPI-SV uses symbolic execution to extract *path-level* models from MPI programs and verifies the models *w.r.t.* the expected properties by model checking [17]. The two techniques complement each other: (1) symbolic execution abstracts the control and data

dependencies to generate verifiable models for model checking, and (2) model checking improves the scalability of symbolic execution by leveraging the verification results to prune redundant paths and enlarges the scope of verifiable properties of symbolic execution.

In particular, MPI-SV combines two algorithms: (1) symbolic execution of *non-blocking* MPI programs with *non-deterministic* operations, and (2) modeling and checking the behaviors of an MPI program path precisely. To safely handle non-deterministic operations, the first algorithm delays the message matchings of non-deterministic operations as much as possible. The second algorithm extracts a model from an MPI program path. The model represents all the path's equivalent behaviors, *i.e.*, the paths generated by changing the interleavings and matchings of the communication operations in the path. We have proved that our modeling algorithm is precise and consistent with the MPI standard [24]. We feed the generated models from the second algorithm into a model checker to perform verification *w.r.t.* the expected properties, *i.e.*, *safety* and *liveness* properties in linear temporal logic (LTL) [58]. If the extracted model from a path  $p$  satisfies the property  $\varphi$ ,  $p$ 's equivalent paths can be safely pruned; otherwise, if the model checker reports a counterexample, a violation of  $\varphi$  is found. This way, we significantly boost the performance of symbolic execution by pruning a large set of paths which are equivalent to certain paths that have been already model-checked.

We have implemented MPI-SV for MPI C programs based on Cloud9 [10] and PAT [80]. We have used MPI-SV to analyze 12 real-world MPI programs, totaling 47K lines of code (LOC) (three are beyond the scale that the state-of-the-art MPI verification tools can handle), *w.r.t.* the deadlock freedom property and *non-reachability* properties. For the 111 deadlock freedom verification tasks, when we set the time threshold to be an hour, MPI-SV can complete 100 tasks, *i.e.*, deadlock reported or deadlock freedom verified, while pure symbolic execution can complete 61 tasks. For the 100 completed tasks, MPI-SV achieves, on average, 19x speedups on verifying deadlock freedom and 5x speedups on finding a deadlock.

The main contributions of this work are:

- A synergistic framework combining symbolic execution and model checking for verifying MPI programs.
- A method for symbolic execution of non-blocking MPI programs with non-deterministic operations. The method is formally proven to preserve the correctness of verifying reachability properties.
- A precise method for modeling the equivalent behaviors of an MPI path, which enlarges the scope of the verifiable properties and improves the scalability.
- A tool for symbolic verification of MPI C programs and an extensive evaluation on real-world MPI programs.

## 2 ILLUSTRATION

In this section, we first introduce MPI programs and use an example to illustrate the problem that this work targets. Then, we overview MPI-SV informally by the example.

### 2.1 MPI Syntax and Motivating Example

MPI implementations, such as MPICH [31] and OpenMPI [26], provide the programming interfaces of message passing to support

```

Proc  ::=  var r : T | r := e | Comm | Proc ; Proc |
          if e Proc else Proc | while e do Proc
Comm  ::=  Ssend(e) | Send(e) | Recv(e) | Recv(*) | Barrier |
          ISend(e,r) | IRecv(e,r) | IRecv(*,r) | Wait(r)

```

Figure 1: Syntax of a core MPI language.

the development of parallel applications. An MPI program can be implemented in different languages, such as C and C++. Without loss of generality, we focus on MPI programs written in C. Let  $\mathbf{T}$  be a set of types,  $\mathbf{N}$  a set of names, and  $\mathbf{E}$  a set of expressions. For simplifying the discussion, we define a core language for MPI processes in Figure 1, where  $\mathbf{T} \in \mathbf{T}$ ,  $\mathbf{r} \in \mathbf{N}$ , and  $\mathbf{e} \in \mathbf{E}$ . An MPI program  $\mathcal{MP}$  is defined by a *finite* set of processes  $\{\text{Proc}_i \mid 0 \leq i \leq n\}$ . *For brevity, we omit complex language features (such as the messages in the communication operations and pointer operations) although MPI-SV does support real-world MPI C programs.*

The statement **var**  $\mathbf{r} : \mathbf{T}$  declares a variable  $\mathbf{r}$  with type  $\mathbf{T}$ . The statement  $\mathbf{r} := \mathbf{e}$  assigns the value of expression  $\mathbf{e}$  to variable  $\mathbf{r}$ . A process can be constructed from basic statements by using the composition operations including sequence, condition and loop. For brevity, we incorporate the key message passing operations in the syntax, where  $\mathbf{e}$  indicates the destination process's identifier. These message passing operations can be *blocking* or *non-blocking*. First, we introduce blocking operations:

- **Ssend**( $\mathbf{e}$ ): sends a message to the  $\mathbf{e}$ th process, and the sending process blocks until the message is received by the destination process.
- **Send**( $\mathbf{e}$ ): sends a message to the  $\mathbf{e}$ th process, and the sending process blocks until the message is copied into the system buffer.
- **Recv**( $\mathbf{e}$ ): receives a message from the  $\mathbf{e}$ th process, and the receiving process blocks until the message from the  $\mathbf{e}$ th process is received.
- **Recv**( $\ast$ ): receives a message from *any* process, and the receiving process blocks until a message is received regardless which process sends the message.
- **Barrier**: blocks the process until all the processes have called **Barrier**.
- **Wait**( $\mathbf{r}$ ): the process blocks until the operation indicated by  $\mathbf{r}$  is completed.

A **Recv**( $\ast$ ) operation, called *wildcard receive*, may receive a message from different processes under different runs, resulting in non-determinism. The blocking of a **Send**( $\mathbf{e}$ ) operation depends on the size of the system buffer, which may differ under different MPI implementations. For simplicity, we assume that the size of the system buffer is infinite. Hence, each **Send**( $\mathbf{e}$ ) operation returns *immediately* after being issued. Note that our implementation allows users to configure the buffer size. To improve the performance, the MPI standard provides non-blocking operations to overlap computations and communications.

- **ISend**( $\mathbf{e}, \mathbf{r}$ ): sends a message to the  $\mathbf{e}$ th process, and the operation returns immediately after being issued. The parameter  $\mathbf{r}$  is the handle of the operation.
- **IRecv**( $\mathbf{e}, \mathbf{r}$ ): receives a message from the  $\mathbf{e}$ th process, and the operation returns immediately after being issued. **IRecv**( $\ast, \mathbf{r}$ ) is the non-blocking wildcard receive.

$P_0$	$P_1$	$P_2$	$P_3$
Send(1)	if ( $x \neq 'a'$ ) Recv(0) else IRecv(*, req); Recv(3)	Send(1)	Send(1)

Figure 2: An illustrative example of MPI programs.

The operations above are key MPI operations. Complex operations, such as MPI\_Bcast and MPI\_Gather, can be implemented by composing these key operations. The formal semantics of the core language is defined based on communicating state machines (CSM) [8]. We define each process as a CSM with an unbounded receiving FIFO queue. For the sake of space limit, the formal semantics can be referred to [91].

An MPI program runs in many processes spanned across multiple machines. These processes communicate by message passing to accomplish a parallel task. Besides parallel execution, the non-determinism in MPI programs mainly comes from two sources: (1) inputs, which may influence the communication through control flow, and (2) wildcard receives, which lead to highly non-deterministic executions.

Consider the MPI program in Figure 2. Processes  $P_0$ ,  $P_2$  and  $P_3$  only send a message to  $P_1$  and then terminate. For process  $P_1$ , if input  $x$  is *not* equal to 'a',  $P_1$  receives a message from  $P_0$  in a blocking manner; otherwise,  $P_1$  uses a non-blocking wildcard receive to receive a message. Then,  $P_1$  receives a message from  $P_3$ . When  $x$  is 'a' and IRecv(\*, req) receives the message from  $P_3$ , a *deadlock* occurs, i.e.,  $P_1$  blocks at Recv(3), and all the other processes terminate. Hence, to detect the deadlock, we need to handle the non-determinism caused by the input  $x$  and the wildcard receive IRecv(\*, req).

To handle non-determinism due to the input, a standard remedy is symbolic execution [48]. However, there are two challenges. The first one is to *systematically explore the paths of an MPI program with non-blocking and wildcard operations*, which significantly increase the complexity of MPI programs. A non-blocking operation does not block but returns immediately, causing out-of-order completion. The difficulty in handling wildcard operations is to get all the possibly matched messages. The second one is to *improve the scalability of the symbolic execution*. Symbolic execution struggles with path explosion. MPI processes run concurrently, resulting in an exponential number of program paths *w.r.t.* the number of processes. Furthermore, the path space increases exponentially with the number of wildcard operations.

## 2.2 Our Approach

MPI-SV leverages dynamic verification [83] and model checking [17] to tackle the challenges. Figure 3 shows MPI-SV's basic framework. The inputs of MPI-SV are an MPI program and an expected property, e.g., *deadlock freedom* expressed in LTL. MPI-SV uses the built-in symbolic executor to explore the path space automatically and checks the property along with path exploration. For a path that violates the property, called a *violation path*, MPI-SV generates a test case for replaying, which includes the program inputs, the

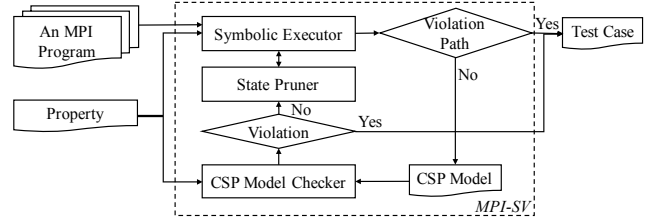


Figure 3: The framework of MPI-SV.

interleaving sequence of MPI operations and the matchings of wildcard receives. In contrast, for a *violation-free* path  $p$ , MPI-SV builds a communicating sequential process (CSP) model  $\Gamma$ , which represents the paths which can be obtained based on  $p$  by changing the interleavings and matchings of the communication operations in  $p$ . Then, MPI-SV utilizes a CSP model checker to verify  $\Gamma$  *w.r.t.* the property. If the model checker reports a counterexample, a violation is found; otherwise, if  $\Gamma$  satisfies the property, MPI-SV prunes all behaviors captured by the model so that they are avoided by symbolic execution.

Since MPI processes are memory independent, MPI-SV will select a process to execute in a *round-robin* manner to avoid exploring all interleavings of the processes. A process keeps running until it *blocks* or *terminates*. When encountering an MPI operation, MPI-SV records the operation instead of executing it and doing the message matching. When every process blocks or terminates and at least one blocked process exists, MPI-SV matches the recorded MPI operations of the processes *w.r.t.* the MPI standard [24]. The intuition behind this strategy is to collect the message exchanges as thoroughly as possible, which helps find possible matchings for the wildcard receive operations. Consider the MPI program in Figure 2 and the *deadlock freedom* property. Figure 4 shows the symbolic execution tree, where the node labels indicate process communications, e.g., (3, 1) means that  $P_1$  receives a message from  $P_3$ . MPI-SV first symbolically executes  $P_0$ , which only sends a message to  $P_1$ . The Send(1) operation returns immediately with the assumption of infinite system buffers. Hence,  $P_0$  terminates, and the operation Send(1) is recorded. Then, MPI-SV executes  $P_1$  and explores both branches of the conditional statement as follows.

(1) **True branch ( $x \neq 'a'$ ).** In this case,  $P_1$  blocks at Recv(0). MPI-SV records the receive operation for  $P_1$ , and starts executing  $P_2$ . Like  $P_0$ ,  $P_2$  executes operation Send(1) and terminates, after which  $P_3$  is selected and behaves the same as  $P_2$ . After  $P_3$  terminates, the global execution blocks, i.e.,  $P_1$  blocks and all the other processes

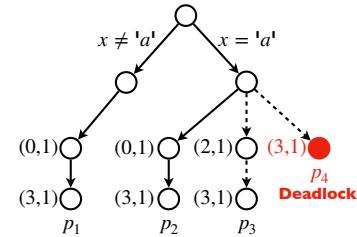


Figure 4: The example program's symbolic execution tree.

terminate. When this happens, MPI-SV matches the recorded operations, performs the message exchanges and continues to execute the matched processes. The  $\text{Recv}(0)$  in  $P_1$  should be matched with the  $\text{Send}(1)$  in  $P_0$ . After executing the send and receive operations, MPI-SV selects  $P_1$  to execute, because  $P_0$  terminates. Then,  $P_1$  blocks at  $\text{Recv}(3)$ . Same as earlier, the global execution blocks and operation matching needs to be done.  $\text{Recv}(3)$  is matched with the  $\text{Send}(1)$  in  $P_3$ . After executing the  $\text{Recv}(3)$  and  $\text{Send}(1)$  operations, all the processes terminate successfully. Path  $p_1$  in Figure 4 is explored.

**(2) False branch ( $x = 'a'$ ).** The execution of  $P_1$  proceeds until reaching the blocking receive  $\text{Recv}(3)$ . Additionally, the two issued receive operations, i.e.,  $\text{IRecv}(*, \text{req})$  and  $\text{Recv}(3)$ , are recorded. Similar to the true branch, when every process blocks or terminates, we handle operation matching. Here  $P_0$ ,  $P_2$  and  $P_3$  terminate, and  $P_1$  blocks at  $\text{Recv}(3)$ .  $\text{IRecv}(*, \text{req})$  should be matched first because of the *non-overtaken* policy in the MPI standard [24]. There are three  $\text{Send}$  operation candidates from  $P_0$ ,  $P_2$  and  $P_3$ , respectively. MPI-SV forks a state for each candidate. Suppose MPI-SV first explores the state where  $\text{IRecv}(*, \text{req})$  is matched with  $P_0$ 's  $\text{Send}(1)$ . After matching and executing  $P_1$ 's  $\text{Recv}(3)$  and  $P_3$ 's  $\text{Send}(1)$ , the path terminates successfully, which generates path  $p_2$  in Figure 4.

**Violation detection.** MPI-SV continues to explore the remaining two cases. Without CSP-based boosting, the deadlock would be found in the last case (i.e.,  $p_4$  in Figure 4), where  $\text{IRecv}(*, \text{req})$  is matched with  $P_3$ 's  $\text{Send}(1)$  and  $P_1$  blocks because  $\text{Recv}(3)$  has no matched operation. MPI-SV generates a CSP model  $\Gamma$  based on the deadlock-free path  $p_2$  where  $P_1$ 's  $\text{IRecv}(*, \text{req})$  is matched with  $P_0$ 's  $\text{Send}(1)$ . Each MPI process is modeled as a CSP process, and all the CSP processes are composed in parallel to form  $\Gamma$ . Notably, in  $\Gamma$ , we collect the possible matchings of a wildcard receive through statically matching the arguments of operations in the path. Additionally, the requirements in the MPI standard, i.e., *completes-before* relations [83], are also modeled. A CSP model checker then verifies deadlock freedom for  $\Gamma$ . The model checker reports a counterexample where  $\text{IRecv}(*, \text{req})$  is matched with the  $\text{Send}(1)$  in  $P_3$ . MPI-SV only explores *two* paths for detecting the deadlock and avoids the exploration of  $p_3$  and  $p_4$  (indicated by dashed lines).

**Pruning.** Because the CSP modeling is precise (cf. Section 4), in addition to finding violations earlier, MPI-SV can also perform path pruning when the model satisfies the property. Suppose we change the program in Figure 2 to be the one where the last statement of  $P_1$  is a  $\text{Recv}(*)$  operation. Then, the program is *deadlock free*. The true branch ( $x \neq 'a'$ ) has 2 paths, because the last wildcard receive in  $P_1$  has two matchings (i.e.,  $P_2$ 's send and  $P_3$ 's send, and  $P_0$ 's send has been matched by  $P_1$ 's  $\text{Recv}(0)$ ). The false branch ( $x = 'a'$ ) has 6 paths because the first wildcard receive has 3 matchings (send operations from  $P_0$ ,  $P_2$  and  $P_3$ ) and the last wildcard receive has 2 matchings (because the first wildcard receive has matched one send operation). Hence, in total, there are 8 paths (i.e.,  $2 + 3 * 2 = 8$ ) if we use pure symbolic execution. In contrast, with model checking, MPI-SV only needs 2 paths to verify that the program is deadlock-free. For each branch, the generated model is verified to be deadlock-free, so MPI-SV prunes the candidate states forked for the matchings of the wildcard receives.

**Properties.** Because our CSP modeling encodes the interleavings of the MPI operations in the MPI processes, the scope of the verifiable properties is enlarged, i.e., MPI-SV can verify safety and liveness properties in LTL. Suppose we change the property to be the one that requires the  $\text{Send}(1)$  operation in  $P_0$  should be completed before the  $\text{Send}(1)$  operation in  $P_2$ . Actually, the send operation in  $P_2$  can be completed before the send operation in  $P_0$ , due to the nature of parallel execution. However, *pure* symbolic execution fails to detect the property violation. In contrast, with the help of CSP modeling, when we verify the model generated from the first path *w.r.t.* the property, the model checker gives a counterexample, indicating that a violation of the property exists.

### 3 SYMBOLIC VERIFICATION METHOD

In this section, we present our symbolic verification framework and then describe MPI-SV's symbolic execution method.

#### 3.1 Framework

Given an MPI program  $\mathcal{MP} = \{\text{Proc}_i \mid 0 \leq i \leq n\}$ , a state  $S_c$  in  $\mathcal{MP}$ 's symbolic execution is composed by the states of processes, i.e.,  $(s_0, \dots, s_n)$ , and each MPI process's state is a 6-tuple  $(\mathcal{M}, \text{Stat}, \text{PC}, \mathcal{F}, \mathcal{B}, \mathcal{R})$ , where  $\mathcal{M}$  maps each variable to a concrete value or a symbolic value,  $\text{Stat}$  is the next program statement to execute,  $\text{PC}$  is the process's path constraint [48],  $\mathcal{F}$  is the flag of process status belonging to  $\{\text{active}, \text{blocked}, \text{terminated}\}$ ,  $\mathcal{B}$  and  $\mathcal{R}$  are infinite buffers for storing the issued MPI operations not yet matched and the matched MPI operations, respectively. We use  $s_i \in S_c$  to denote that  $s_i$  is a process state in the global state  $S_c$ . An element *elem* of  $s_i$  can be accessed by  $s_i.\text{elem}$ , e.g.,  $s_i.\mathcal{F}$  is the  $i$ th process's status flag. In principle, a statement execution in any process advances the global state, making  $\mathcal{MP}$ 's state space exponential to the number of processes. We use variable  $\text{Seq}_i$  defined in  $\mathcal{M}$  to record the sequence of the issued MPI operations in  $\text{Proc}_i$ , and  $\text{Seq}(S_c)$  to denote the set  $\{\text{Seq}_i \mid 0 \leq i \leq n\}$  of global state  $S_c$ . Global state  $S_c$ 's path condition (denoted by  $S_c.\text{PC}$ ) is the conjunction of the path conditions of  $S_c$ 's processes, i.e.,  $\bigwedge_{s_i \in S_c} s_i.\text{PC}$ .

Algorithm 1 shows the details of MPI-SV. We use *worklist* to store the global states to be explored. Initially, *worklist* only contains  $S_{\text{init}}$ , composed of the initial states of all the processes, and each process's status is active. At Line 4, Select picks a state from *worklist* as the one to advance. Hence, Select can be customized with different search heuristics, e.g., depth-first search (DFS). Then, Scheduler selects an active process  $\text{Proc}_i$  to execute. Next, Execute (cf. Algorithm 2) symbolically executes the statement  $\text{Stat}_i$  in  $\text{Proc}_i$ , and may add new states into *worklist*. This procedure continues until *worklist* is empty (i.e., all the paths have been explored), detecting a violation or time out (omitted for brevity). After executing  $\text{Stat}_i$ , if all the processes in the current global state  $S_c$  terminate, i.e., a violation-free path terminates, we use Algorithm 4 to generate a CSP model  $\Gamma$  from the current state (Line 8). Then, we use a CSP model checker to verify  $\Gamma$  *w.r.t.*  $\varphi$ . If  $\Gamma$  satisfies  $\varphi$  (denoted by  $\Gamma \models \varphi$ ), we prune the global states forked by the wildcard operations along the current path (Line 11), i.e., the states in *worklist* whose path conditions imply  $S_c$ 's path condition; otherwise, if the model checker gives a counterexample, we report the violation and exit (Line 14).

**Algorithm 1:** Symbolic Verification Framework

---

```

MPI-SV( $\mathcal{MP}, \varphi, \text{Sym}$ )
Data:  $\mathcal{MP}$  is  $\{\text{Proc}_i \mid 0 \leq i \leq n\}$ ,  $\varphi$  is a property, and Sym
is a set of symbolic variables
1 begin
2    $\text{worklist} \leftarrow \{S_{\text{init}}\}$ 
3   while  $\text{worklist} \neq \emptyset$  do
4      $S_c \leftarrow \text{Select}(\text{worklist})$ 
5      $(\mathcal{M}_i, \text{Stat}_i, \text{PC}_i, \mathcal{F}_i, \mathcal{B}_i, \mathcal{R}_i) \leftarrow \text{Scheduler}(S_c)$ 
6      $\text{Execute}(S_c, \text{Proc}_i, \text{Stat}_i, \text{Sym}, \text{worklist})$ 
7     if  $\forall s_i \in S_c, s_i.\mathcal{F} = \text{terminated}$  then
8        $\Gamma \leftarrow \text{GenerateCSP}(S_c)$ 
9        $\text{ModelCheck}(\Gamma, \varphi)$ 
10      if  $\Gamma \models \varphi$  then
11         $\text{worklist} \leftarrow \text{worklist} \setminus \{S_p \in \text{worklist} \mid S_p.PC \Rightarrow S_c.PC\}$ 
12      end
13      else if  $\Gamma \not\models \varphi$  then
14         $\text{reportViolation and Exit}$ 
15      end
16    end
17  end
18 end

```

---

Since MPI processes are memory independent, we employ partial order reduction (POR) [17] to reduce the search space. Scheduler selects a process in a *round-robin* fashion from the current global state. In principle, Scheduler starts from the active MPI process with the smallest identifier, e.g.,  $\text{Proc}_0$  at the beginning, and an MPI process keeps running until it is blocked or terminated. Then, the next active process will be selected to execute. Such a strategy significantly reduces the path space of symbolic execution. Then, with the help of CSP modeling and model checking, MPI-SV can verify more properties, i.e., safety and liveness properties in LTL. The details of such technical improvements will be given in Section 4.

### 3.2 Blocking-driven Symbolic Execution

Algorithm 2 shows the symbolic execution of a statement. Common statements such as conditional statements are handled in the standard way [48] (omitted for brevity), and here we focus on MPI operations. The main idea is to *delay* the executions of MPI operations *as much as possible*, i.e., trying to get all the message matchings. Instead of execution, Algorithm 2 records each MPI operation for each MPI process (Lines 4&8). We also need to update buffer  $\mathcal{B}$  after issuing an MPI operation (Lines 5&9). Then, if  $\text{Stat}_i$  is a non-blocking operation, the execution returns immediately; otherwise, we block  $\text{Proc}_i$  (Line 10, excepting the Wait of an ISend operation). When reaching GlobalBlocking (Lines 11&12), i.e., every process is terminated or blocked, we use Matching (cf. Algorithm 3) to match the recorded but not yet matched MPI operations and execute the matched operations. Since the opportunity of matching messages is GlobalBlocking, we call it blocking-driven symbolic execution.

Matching matches the recorded MPI operations in different processes. To obtain all the possible matchings, we delay the matching of a wildcard operation *as much as possible*. We use  $\text{match}_N$  to match the non-wildcard operations first (Line 3) w.r.t. the rules in the MPI standard [24], especially the *non-overtaken* ones: (1) if two sends of a process send messages to the same destination, and both

**Algorithm 2:** Blocking-driven Symbolic Execution

---

```

Execute( $S_c, \text{Proc}_i, \text{Stat}_i, \text{Sym}, \text{worklist}$ )
Data: Global state  $S_c$ , MPI process  $\text{Proc}_i$ , Statement  $\text{Stat}_i$ ,
Symbolic variable set Sym,  $\text{worklist}$  of global states
1 begin
2   switch ( $\text{Stat}_i$ ) do
3     case Send or ISend or IRecv do
4        $\text{Seq}_i \leftarrow \text{Seq}_i \cdot \langle \text{Stat}_i \rangle$ 
5        $s_i.\mathcal{B} \leftarrow s_i.\mathcal{B} \cdot \langle \text{Stat}_i \rangle$ 
6     end
7     case Barrier or Wait or Ssend or Recv do
8        $\text{Seq}_i \leftarrow \text{Seq}_i \cdot \langle \text{Stat}_i \rangle$ 
9        $s_i.\mathcal{B} \leftarrow s_i.\mathcal{B} \cdot \langle \text{Stat}_i \rangle$ 
10       $s_i.\mathcal{F} \leftarrow \text{blocked}$ 
11      if GlobalBlocking then
12        //  $\forall s_i \in S_c, (s_i.\mathcal{F} = \text{blocked} \vee s_i.\mathcal{F} = \text{terminated})$ 
13        Matching( $S_c, \text{worklist}$ )
14      end
15      default:
16        Execute( $S_c, \text{Proc}_i, \text{Stat}_i, \text{Sym}, \text{worklist}$ ) as normal
17  end

```

---

can match the same receive, the receive should match the first one; and (2) if a process has two receives, and both can match a send, the first receive should match the send. The matched send and receive operations will be executed, and the statuses of the involved processes will be updated to active, denoted by  $\text{Fire}(S_c, \text{pair}_n)$  (Line 5). If there is no matching for non-wildcard operations, we use  $\text{match}_W$  to match the wildcard operations (Line 8). For each possible matching of a wildcard receive, we fork a new state (denoted by  $\text{fork}(S_c, \text{pair}_w)$  at Line 10) to analyze each matching case. If no operations can be matched, but there exist blocked processes, a

**Algorithm 3:** Blocking-driven Matching

---

```

Matching( $S_c, \text{worklist}$ )
Data: Global state  $S_c$ ,  $\text{worklist}$  of global states
1 begin
2    $MS_W \leftarrow \emptyset$  // Matching set of wildcard operations
3    $\text{pair}_n \leftarrow \text{match}_N(S_c)$  // Match non-wildcard operations
4   if  $\text{pair}_n \neq \text{empty pair}$  then
5      $\text{Fire}(S_c, \text{pair}_n)$ 
6   end
7   else
8      $MS_W \leftarrow \text{match}_W(S_c)$  // Match wildcard operations
9     for  $\text{pair}_w \in MS_W$  do
10       $S'_c \leftarrow \text{fork}(S_c, \text{pair}_w)$ 
11       $\text{worklist} \leftarrow \text{worklist} \cup \{S'_c\}$ 
12    end
13    if  $MS_W \neq \emptyset$  then
14       $\text{worklist} \leftarrow \text{worklist} \setminus \{S_c\}$ 
15    end
16  end
17  if  $\text{pair}_n = \text{empty pair} \wedge MS_W = \emptyset$  then
18     $\text{reportDeadlock and Exit}$ 
19  end
20 end

```

---

$P_0$	$P_1$	$P_2$
ISend(1, req <sub>1</sub> );	IRecv(*, req <sub>2</sub> );	Barrier;
Barrier;	Barrier;	ISend(1, req <sub>3</sub> );
Wait(req <sub>1</sub> )	Wait(req <sub>2</sub> )	Wait(req <sub>3</sub> )

Figure 5: An example of operation matching.

deadlock happens (Line 17). Besides, for the LTL properties other than deadlock freedom (such as temporal properties), we also check them during symbolic execution (omitted for brevity).

Take the program in Figure 5 for example. When all the processes block at Barrier, MPI-SV matches the recorded operation in the buffers of the processes, i.e.,  $s_0.B = \langle \text{ISend}(1, \text{req}_1), \text{Barrier} \rangle$ ,  $s_1.B = \langle \text{IRecv}(*, \text{req}_2), \text{Barrier} \rangle$ , and  $s_2.B = \langle \text{Barrier} \rangle$ . According to the MPI standard, each operation in the buffers is ready to be matched. Hence, Matching first matches the non-wildcard operations, i.e., the Barrier operations, then the status of each process becomes active. After that, MPI-SV continues to execute the active processes and record issued MPI operations. The next GlobalBlocking point is:  $P_0$  and  $P_2$  terminate, and  $P_1$  blocks at Wait(req<sub>2</sub>). The buffers are  $\langle \text{ISend}(1, \text{req}_1), \text{Wait}(\text{req}_1) \rangle$ ,  $\langle \text{IRecv}(*, \text{req}_2), \text{Wait}(\text{req}_2) \rangle$ , and  $\langle \text{ISend}(1, \text{req}_3), \text{Wait}(\text{req}_3) \rangle$ , respectively. All the issued Wait operations are not ready to match, because the corresponding non-blocking operations are not matched. So Matching needs to match the wildcard operation, i.e., IRecv(\*, req<sub>2</sub>), which can be matched with ISend(1, req<sub>1</sub>) or ISend(1, req<sub>3</sub>). Then, a new state is forked for each case and added to the *worklist*.

**Correctness.** Blocking-driven symbolic execution is an instance of model checking with POR. We have proved the symbolic execution method is correct for *reachability properties* [58]. Due to the space limit, the proof can be referred to [91].

## 4 CSP BASED PATH MODELING

In this section, we first introduce the CSP [70] language. Then, we present the modeling algorithm of an MPI program terminated path using a subset of CSP. Finally, we prove the soundness and completeness of our modeling.

### 4.1 CSP Subset

Let  $\Sigma$  be a *finite* set of *events*,  $\mathbb{C}$  a set of channels, and  $X$  a set of variables. Figure 6 shows the syntax of the CSP subset, where  $P$  denotes a CSP process,  $a \in \Sigma$ ,  $c \in \mathbb{C}$ ,  $X \subseteq \Sigma$  and  $x \in X$ .

$$P := a \mid P \circ P \mid P \sqcap P \mid P \parallel_X P \mid c?x \rightarrow P \mid c!x \rightarrow P \mid \text{skip}$$

Figure 6: The syntax of a CSP subset.

The single event process  $a$  performs the event  $a$  and terminates. There are three operators: sequential composition ( $\circ$ ), external choice ( $\sqcap$ ) and parallel composition with synchronization ( $\parallel_X$ ).  $P \sqcap Q$  performs as  $P$  or  $Q$ , and the choice is made by the environment. Let  $PS$  be a finite set of processes,  $\sqcap PS$  denotes the external choice of all the processes in  $PS$ .  $P \parallel_X Q$  performs  $P$  and  $Q$  in an interleaving manner, but  $P$  and  $Q$  synchronize on the events in  $X$ . The

process  $c?x \rightarrow P$  performs as  $P$  after reading a value from channel  $c$  and writing the value to variable  $x$ . The process  $c!x \rightarrow P$  writes the value of  $x$  to channel  $c$  and then behaves as  $P$ . Process **skip** terminates immediately.

### 4.2 CSP Modeling

For each violation-free program path, Algorithm 4 builds a precise CSP model of the possible communication behaviors by changing the matchings and interleavings of the communication operations along the path. The basic idea is to model the communication operations in each process as a CSP process, then compose all the CSP processes in parallel to form the model. To model Proc<sub>*i*</sub>, we

---

#### Algorithm 4: CSP Modeling for a Terminated State

---

```

GenerateCSP(S)
Data: A terminated global state  $S$ , and
         $\text{Seq}(S) = \{\text{Seq}_i \mid 0 \leq i \leq n\}$ 
1 begin
2    $PS \leftarrow \emptyset$ 
3   for  $i \leftarrow 0 \dots n$  do
4      $P_i \leftarrow \text{skip}$ 
5      $\text{Req} \leftarrow \{r \mid \text{IRecv}(*, r) \in \text{Seq}_i \vee \text{IRecv}(i, r) \in \text{Seq}_i\}$ 
6     for  $j \leftarrow \text{length}(\text{Seq}_i) - 1 \dots 0$  do
7       switch  $op_j$  do
8         case Ssend( $i$ ) do
9            $c_1 \leftarrow \text{Chan}(op_j)$  //  $c_1$ 's size is 0
10           $P_i \leftarrow c_1!x \rightarrow P_i$ 
11        end
12        case Send( $i$ ) or ISend( $i, r$ ) do
13           $c_2 \leftarrow \text{Chan}(op_j)$  //  $c_2$ 's size is 1
14           $P_i \leftarrow c_2!x \rightarrow P_i$ 
15        end
16        case Barrier do
17           $P_i \leftarrow B \circ P_i$ 
18        end
19        case Recv( $i$ ) or Recv( $*$ ) do
20           $C \leftarrow \text{StaticMatchedChannel}(op_j, S)$ 
21           $Q \leftarrow \text{Refine}(\sqcap \{c?x \rightarrow \text{skip} \mid c \in C\}, S)$ 
22           $P_i \leftarrow Q \circ P_i$ 
23        end
24        case IRecv( $*$ ,  $r$ ) or IRecv( $i, r$ ) do
25           $C \leftarrow \text{StaticMatchedChannel}(op_j, S)$ 
26           $Q \leftarrow \text{Refine}(\sqcap \{c?x \rightarrow \text{skip} \mid c \in C\}, S)$ 
27           $e_w \leftarrow \text{WaitEvent}(op_j)$  //  $op_j$ 's wait event
28           $P_i \leftarrow (Q \circ e_w) \parallel_{\{e_w\}} P_i$ 
29        end
30        case Wait( $r$ ) and  $r \in \text{Req}$  do
31           $e_w \leftarrow \text{GenerateEvent}(op_j)$ 
32           $P_i \leftarrow e_w \circ P_i$ 
33        end
34      end
35    end
36     $PS \leftarrow PS \cup \{P_i\}$ 
37  end
38   $P \leftarrow \parallel_{\{B\}} PS$ 
39  return  $P$ 
40 end

```

---

scan its operation sequence  $Seq_i$  in reverse. For each operation, we generate its CSP model and compose the model with that of the remaining operations in  $Seq_i$  w.r.t. the semantics of the operation and the MPI standard [24]. The modeling algorithm is efficient, and has a polynomial time complexity w.r.t. the total length of the recorded MPI operation sequences.

We use channel operations in CSP to model send and receive operations. Each send operation  $op$  has its own channel, denoted by  $Chan(op)$ . We use a *zero-sized* channel to model Ssend operation (Line 10), because Ssend blocks until the message is received. In contrast, considering a Send or ISend operation is completed immediately, we use *one-sized* channels for them (Line 14), so the channel writing returns immediately. The modeling of Barrier (Line 17) is to generate a synchronization event that requires all the parallel CSP processes to synchronize it (Lines 17&38). The modeling of receive operations consists of three steps. The first step calculates the possibly matched channels written by the send operations (Lines 20&25). The second uses the external choice of reading actions of the matched channels (Lines 21&26), so as to model different cases of the receive operation. Finally, the refined external choice process is composed with the remaining model. If the operation is blocking, the composition is sequential (Line 22); otherwise, it is a parallel composition (Line 28).

StaticMatchedChannel( $op_j, S$ ) (Lines 20&25) returns the set of the channels written by the possibly matched send operations of the receive operation  $op_j$ . We scan  $Seq(S)$  to obtain the possibly matched send operations of  $op_j$ . Given a receive operation  $recv$  in process  $Proc_i$ ,  $SMO(recv, S)$  calculated as follows denotes the set of the matched send operations of  $recv$ .

- If  $recv$  is  $Recv(j)$  or  $IRecv(j, r)$ ,  $SMO(recv, S)$  contains  $Proc_j$ 's send operations with  $Proc_i$  as the destination process.
- If  $recv$  is  $Recv(*)$  or  $IRecv(*, r)$ ,  $SMO(recv, S)$  contains *any* process's send operations with  $Proc_i$  as the destination process.

$SMO(op, S)$  over-approximates  $op$ 's precisely matched operations, and can be optimized by removing the send operations that are definitely executed after  $op$ 's completion, and the ones whose messages are definitely received before  $op$ 's issue. For example, Let  $Proc_0$  be  $Send(1);Barrier;Send(1)$ , and  $Proc_1$  be  $Recv(*);Barrier$ .  $SMO$  will add the two send operations in  $Proc_0$  to the matching set of the  $Recv(*)$  in  $Proc_1$ . Since  $Recv(*)$  must complete before  $Barrier$ , we can remove the second send operation in  $Proc_0$ . Such optimization reduces the complexity of the CSP model. For brevity, we use  $SMO(op, S)$  to denote the optimized matching set. Then,  $StaticMatchedChannel(op_j, S)$  is  $\{Chan(op) \mid op \in SMO(op_j, S)\}$ .

To satisfy the MPI requirements,  $Refine(P, S)$  (Lines 21&26) refines the models of receive operations by imposing the completes-before requirements [83] as follows:

- If a receive operation has multiple matched send operations from the same process, it should match the earlier issued one. This is ensured by checking the emptiness of the dependent channels.
- The receive operations in the same process should be matched w.r.t. their issue order if they receive messages from the same process, except the *conditional completes-before* pattern [83]. We use one-sized channel actions to model these requirements.

We model a Wait operation if it corresponds to an IRecv operation (Line 30), because ISend operations complete immediately

under the assumption of infinite system buffer. Wait operations are modeled by the synchronization in parallel processes. GenerateEvent generates a new synchronization event  $e_w$  for each Wait operation (Line 31). Then,  $e_w$  is produced after the corresponding non-blocking operation is completed (Line 28). The synchronization on  $e_w$  ensures that a Wait operation blocks until the corresponding non-blocking operation is completed.

We use the example in Figure 5 for a demonstration. After exploring a violation-free path, the recorded operation sequences are  $Seq_0 = \langle ISend(1, req_1), Barrier, Wait(req_1) \rangle$ ,  $Seq_1 = \langle IRecv(*, req_2), Barrier, Wait(req_2) \rangle$ ,  $Seq_2 = \langle Barrier, ISend(1, req_3), Wait(req_3) \rangle$ . We first scan  $Seq_0$  in reverse. Note that we don't model  $Wait(req_1)$ , because it corresponds to  $ISend$ . We create a synchronization event  $B$  for modeling  $Barrier$  (Lines 16&17). For the  $ISend(1, req_1)$ , we model it by writing an element  $a$  to a one-sized channel  $chan_1$ , and use prefix operation to compose its model with  $B$  (Lines 12-14). In this way, we generate CSP process  $chan_1!a \rightarrow B \S skip$  (denoted by  $CP_0$ ) for  $Proc_0$ . Similarly, we model  $Proc_2$  by  $B \S chan_2!b \rightarrow skip$  (denoted by  $CP_2$ ), where  $chan_2$  is also a one-sized channel and  $b$  is a channel element. For  $Proc_1$ , we generate a single event process  $e_w$  to model  $Wait(req_2)$ , because it corresponds to  $IRecv$  (Lines 30-32). For  $IRecv(*, req_2)$ , we first compute the matched channels using  $SMO$  (Line 25), and  $StaticMatchedChannel(op_j, S)$  contains both  $chan_1$  and  $chan_2$ . Then, we generate the following CSP process

$$((chan_1?a \rightarrow skip \sqcap chan_2?b \rightarrow skip) \S e_w) \parallel (B \S e_w \S skip)_{\{e_w\}}$$

(denoted by  $CP_1$ ) for  $Proc_1$ . Finally, we compose the CSP processes using the parallel operator to form the CSP model (Line 38), i.e.,  $CP_0 \parallel_{\{B\}} CP_1 \parallel_{\{B\}} CP_2$ .

CSP modeling supports the case where communications depend on message contents. MPI-SV tracks the influence of a message during symbolic execution. When detecting that the message content influences the communications, MPI-SV symbolizes the content on-the-fly. We specially handle the widely used *master-slave* pattern for dynamic load balancing [32]. The basic idea is to use a recursive CSP process to model each slave process and a conditional statement for master process to model the communication behaviors of different matchings. We verified five dynamic load balancing MPI programs in our experiments (cf. Section 5.4). The details for supporting master-slave pattern is in the supplementary document.

### 4.3 Soundness and Completeness

In the following, we show that the CSP modeling is *sound* and *complete*. Suppose  $GenerateCSP(S)$  generates the CSP process  $CSP_s$ . Here, *soundness* means that  $CSP_s$  models all the possible behaviors by changing the matchings or interleavings of the communication operations along the path to  $S$ , and *completeness* means that each trace in  $CSP_s$  represents a real behavior that can be derived from  $S$  by changing the matchings or interleavings of the communications.

Since we compute  $SMO(op, S)$  by statically matching the arguments of the recorded operations,  $SMO(op, S)$  may contain some false matchings. Calculating the precisely matched operations of  $op$  is NP-complete [23], and we suppose such an ideal method exists. We use  $CSP_{static}$  and  $CSP_{ideal}$  to denote the generated models



using  $\text{SMO}(op, S)$  and the ideal method, respectively. The following theorems ensure the equivalence of the two models under the stable-failure semantics [70] of CSP and  $\text{CSP}_{\text{static}}$ 's consistency to the MPI semantics, which imply the soundness and completeness of our CSP modeling method. Let  $\mathcal{T}(P)$  denote the trace set [70] of CSP process  $P$ , and  $\mathcal{F}(P)$  denote the failure set of CSP process  $P$ . Each element in  $\mathcal{F}(P)$  is  $(s, X)$ , where  $s \in \mathcal{T}(P)$  is a trace, and  $X$  is the set of events  $P$  refuses to perform after  $s$ .

**Theorem 4.1.**  $\mathcal{F}(\text{CSP}_{\text{static}}) = \mathcal{F}(\text{CSP}_{\text{ideal}})$ .

PROOF. We only give the skeleton of the proof. We first prove

$$\mathcal{T}(\text{CSP}_{\text{static}}) = \mathcal{T}(\text{CSP}_{\text{ideal}})$$

based on which we can prove  $\mathcal{F}(\text{CSP}_{\text{static}}) = \mathcal{F}(\text{CSP}_{\text{ideal}})$ . The main idea of proving these two equivalence relations is to use *contradiction* for proving the subset relations. We only give the proof of  $\mathcal{T}(\text{CSP}_{\text{static}}) \subseteq \mathcal{T}(\text{CSP}_{\text{ideal}})$ ; the other subset relations can be proved in a similar way.

Suppose there is a trace  $t = \langle e_1, \dots, e_n \rangle$  such that  $t \in \mathcal{T}(\text{CSP}_{\text{static}})$  but  $t \notin \mathcal{T}(\text{CSP}_{\text{ideal}})$ . The only difference between  $\text{CSP}_{\text{static}}$  and  $\text{CSP}_{\text{ideal}}$  is that  $\text{CSP}_{\text{static}}$  introduces more channel read operations during the modeling of receive operations. Hence, there must exist a read operation of an extra channel in  $t$ . Suppose the first extra read is  $e_k = c_e ? x$ , where  $1 \leq k \leq n$ . Therefore,  $c_e$  cannot be read in  $\text{CSP}_{\text{ideal}}$  when the matching of the corresponding receive operation starts, but  $c_e$  is not empty at  $e_k$  in  $\text{CSP}_{\text{static}}$ . Despite of the size of  $c_e$ , there must exist a write operation  $c_e ! y$  in  $\langle e_1, \dots, e_{k-1} \rangle$ . Because  $\langle e_1, \dots, e_{k-1} \rangle$  is also a valid trace in  $\text{CSP}_{\text{ideal}}$ , it means  $c_e$  is not empty in  $\text{CSP}_{\text{ideal}}$  at  $e_k$ , which contradicts with the assumption that  $c_e$  cannot be read in  $\text{CSP}_{\text{ideal}}$ . Hence,  $\mathcal{T}(\text{CSP}_{\text{static}}) \subseteq \mathcal{T}(\text{CSP}_{\text{ideal}})$  holds.  $\square$

**Theorem 4.2.**  $\text{CSP}_{\text{static}}$  is consistent with the MPI semantics.

The proof's main idea is to prove that  $\text{CSP}_{\text{ideal}}$  is equal to the model defined by the formal MPI semantics [91] w.r.t. the failure divergence semantics. Then, based on Theorem 4.1, we can prove that  $\text{CSP}_{\text{static}}$  is consistent with the MPI semantics. Please refer to [91] for the detailed proofs for these two theorems.

## 5 EXPERIMENTAL EVALUATION

In this section, we first introduce the implementation of MPI-SV, then describes the research questions and the experimental setup. Finally, we give experimental results.

### 5.1 Implementation

We have implemented MPI-SV based on Cloud9 [10], which is built upon KLEE [12], and enhances KLEE with better support for POSIX environment and parallel symbolic execution. We leverage Cloud9's support for multi-threaded programs. We use a multi-threaded library for MPI, called AzequiaMPI [69], as the MPI environment model for symbolic execution. MPI-SV contains three main modules: program preprocessing, symbolic execution, and model checking. The program preprocessing module generates the input for symbolic execution. We use Clang to compile an MPI program to LLVM bytecode, which is then linked with the pre-compiled MPI library AzequiaMPI. The symbolic execution module is in charge of path exploration and property checking. The third module utilizes the

**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
<b>Total</b>	<b>47354</b>	<b>12 open source programs</b>

state-of-the-art CSP model checker PAT [80] to verify CSP models, and uses the output of PAT to boost the symbolic executor.

### 5.2 Research Questions

We conducted experiments to answer the following questions:

- Effectiveness: Can MPI-SV verify real-world MPI programs effectively? How effective is MPI-SV when compared to the existing state-of-the-art tools?
- Efficiency: How efficient is MPI-SV when verifying real-world MPI programs? How efficient is MPI-SV when compared to the pure symbolic execution?
- Verifiable properties : Can MPI-SV verify properties other than deadlock freedom?

### 5.3 Setup

Table 1 lists the programs analyzed in our experiments. All the programs are real-world open source MPI programs. DTG is a testing program from [82]. Matmat, Integrate and Diffusion2d come from the FEVS benchmark suite [76]. Matmat is used for matrix multiplication, Integrate calculates the integrals of trigonometric functions, and Diffusion2d is a parallel solver for two-dimensional diffusion equation. Gauss\_elim is an MPI implementation for gaussian elimination used in [88]. Heat is a parallel solver for heat equation used in [61]. Mandelbrot, Sorting and Image\_manip come from github. Mandelbrot parallel draws the mandelbrot set for a bitmap, Sorting uses bubble sort to sort a multi-dimensional array, and Image\_manip is an MPI program for image manipulations, e.g., shifting, rotating and scaling. The remaining three programs are large parallel applications. Depsolver is a parallel multi-material 3D electrostatic solver, Kfray is a ray tracing program creating realistic images, and ClustalW is a tool for aligning gene sequences.

To evaluate MPI-SV further, we mutate [46] the programs by rewriting a randomly selected receive using two rules: (1) replace  $\text{Recv}(i)$  with  $\text{if } (x > a) \{ \text{Recv}(i) \} \text{ else } \{ \text{Recv}(*) \}$ ; (2) replace  $\text{Recv}(*)$  with  $\text{if } (x > a) \{ \text{Recv}(*) \} \text{ else } \{ \text{Recv}(j) \}$ . Here  $x$  is an input variable,  $a$  is a random value, and  $j$  is generated randomly from the scope of the process identifier. The mutations for  $\text{IRcv}(i, r)$  and  $\text{IRcv}(*, r)$  are similar. Rule 1 is to improve program performance and simplify



programming, while rule 2 is to make the communication more deterministic. Since communications tend to depend on inputs in complex applications, such as the last three programs in Table 1, we also introduce input related conditions. For each program, we generate five mutants if possible, or generate as many as the number of receives. We don't mutate the programs using *master-slave* pattern [32], *i.e.*, *Matmat* and *Sorting*, and only mutate the static scheduling versions of programs *Integrate*, *Mandelbrot*, and *Kfray*.

**Baselines.** We use pure symbolic execution as the first baseline because: (1) none of the state-of-the-art symbolic execution based verification tools can analyze non-blocking MPI programs, *e.g.*, CIVL [57, 75]; (2) MPI-SPIN [74] can support input coverage and non-blocking operations, but it requires building models of the programs manually; and (3) other automated tools that support non-blocking operations, such as MOPPER [23] and ISP [83], can only verify programs under given inputs. MPI-SV aims at covering both the input space and non-determinism automatically. To compare with pure symbolic execution, we run MPI-SV under two configurations: (1) Symbolic execution, *i.e.*, applying only symbolic execution for path exploration, and (2) Our approach, *i.e.*, using model checking based boosting. Most of the programs run with 6, 8, and 10 processes, respectively. DTG and Matmat can only be run with 5 and 4 processes, respectively. For Diffusion and the programs using the *master-slave* pattern, we only run them with 4 and 6 processes due to the huge path space. We use MPI-SV to verify deadlock freedom of MPI programs and also evaluate 2 *non-reachability* properties for *Integrate* and *Mandelbrot*. The timeout is one hour. There are three possible verification results: finding a violation, no violation, or timeout. We carry out all the tasks on an Intel Xeon-based Server with 64G memory and 8 2.5GHz cores running a Ubuntu 14.04 OS. We ran each verification task three times and use the average results to alleviate the experimental errors. To evaluate MPI-SV's effectiveness further, we also directly compare MPI-SV with CIVL [57, 75] and MPI-SPIN [74]. Note that, since MPI-SPIN needs manual modeling, we only use MPI-SV to verify MPI-SPIN's C benchmarks *w.r.t.* deadlock freedom.

## 5.4 Experimental Results

Table 2 lists the results for evaluating MPI-SV against pure symbolic execution. The first column shows program names, and #Procs is the number of running processes. T specifies whether the analyzed program is mutated, where *o* denotes the original program, and *m<sub>i</sub>* represents a mutant. A task comprises a program and the number of running processes. We label the programs using *master-slave* pattern with superscript “\*”. Column **Deadlock** indicates whether a task is deadlock free, where 0, 1, and -1 denote *no deadlock*, *deadlock* and *unknown*, respectively. We use unknown for the case that both configurations fail to complete the task. Columns **Time(s)** and **#Iterations** show the verification time and the number of explored paths, respectively, where TO stands for timeout. The results where Our approach performs better is in gray background.

For the 111 verification tasks, MPI-SV completes 100 tasks (90%) within one hour, whereas 61 tasks (55%) for Symbolic execution. Our approach detects deadlocks in 48 tasks, while the number of Symbolic execution is 44. We manually confirmed that the detected deadlocks are real. For the 48 tasks having deadlocks, MPI-SV on

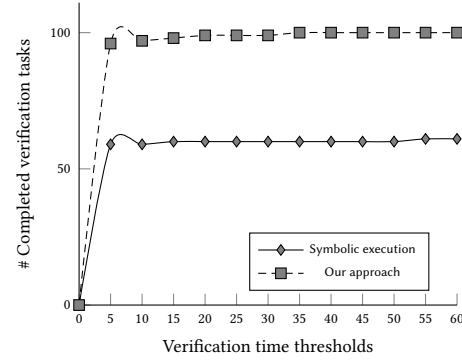


Figure 7: Completed tasks under a time threshold.

average offers a 5x speedups for detecting deadlocks. On the other hand, Our approach can verify deadlock freedom for 52 tasks, while only 17 tasks for Symbolic execution. MPI-SV achieves an average 19x speedups. Besides, compared with Symbolic execution, Our approach requires fewer paths to detect the deadlocks (1/55 on average) and complete the path exploration (1/205 on average). These results demonstrate MPI-SV's effectiveness and efficiency.

Figure 7 shows the efficiency of verification for the two configurations. The X-axis varies the time threshold from 5 minutes to one hour, while the Y-axis is the number of completed verification tasks. Our approach can complete more tasks than Symbolic execution under the same time threshold, demonstrating MPI-SV's efficiency. In addition, Our approach can complete 96 (96%) tasks in 5 minutes, which also demonstrates MPI-SV's effectiveness.

For some tasks, *e.g.*, *Kfray*, MPI-SV does not outperform Symbolic execution. The reasons include: (a) the paths contain hundreds of non-wildcard operations, and the corresponding CSP models are huge, and thus time-consuming to model check; (b) the number of wildcard receives or their possible matchings is very small, and as a result, only few paths are pruned.

**Comparison with CIVL.** CIVL uses symbolic execution to build a model for the whole program and performs model checking on the model. In contrast, MPI-SV adopts symbolic execution to generate *path-level verifiable* models. CIVL does not support non-blocking operations. We applied CIVL on our evaluation subjects. It only successfully analyzed DTG. Diffusion2d could be analyzed after removing unsupported external calls. MPI-SV and CIVL had similar performance on these two programs. CIVL failed on all the remaining programs due to compilation failures or lack of support for non-blocking operations. In contrast, MPI-SV successfully analyzed 99 of the 140 programs in CIVL's latest benchmarks. The failed ones are small API test programs for the APIs that MPI-SV does not support. For the real-world program *floyd* that both MPI-SV and CIVL can analyze, MPI-SV verified its deadlock-freedom under 4 processes in 3 minutes, while CIVL timed out after 30 minutes. The results indicate the benefits of MPI-SV's path-level modeling.

**Comparison with MPI-SPIN.** MPI-SPIN relies on manual modeling of MPI programs. Inconsistencies may happen between an MPI program and its model. Although prototypes exist for translating C to Promela [45], they are impractical for real-world MPI programs. MPI-SPIN's state space reduction treats communication

Table 2: Experimental results.

Program (#Procs)	T	Deadlock	Time(s)		#Iterations	
			Symbolic execution	Our approach	Symbolic execution	Our approach
DTG(5)	<i>o</i>	0	10.12	9.02	3	1
	<i>m</i> <sub>1</sub>	0	13.69	9.50	10	2
	<i>m</i> <sub>2</sub>	1	10.02	8.93	4	2
	<i>m</i> <sub>3</sub>	1	10.21	9.49	4	2
	<i>m</i> <sub>4</sub>	1	10.08	9.19	4	2
	<i>m</i> <sub>5</sub>	1	9.04	9.29	2	2
Matmat*(4)	<i>o</i>	0	36.94	10.43	54	1
Integrate(6/8/10)	<i>o</i>	0/0/0	78.17/TO/TO	8.87/10.45/44.00	120/3912/3162	1/1/1
	<i>m</i> <sub>1</sub>	0/0/-1	TO/TO/TO	49.94 /TO/TO	4773/3712/3206	32 /128/79
	<i>m</i> <sub>2</sub>	1/1/1	9.35/9.83/9.94	9.39/10.76/44.09	2/2/2	2/2/2
Integrate*(4/6)	<i>o</i>	0/0	24.18/123.55	9.39/32.03	27/125	1/1
Diffusion2d(4/6)	<i>o</i>	0/0	106.86/TO	9.84/13.19	90/2041	1/1
	<i>m</i> <sub>1</sub>	0/1	110.25/11.95	10.18 /13.81	90/2	1 /2
	<i>m</i> <sub>2</sub>	0/1	3236.02/12.66	17.05 /14.38	5850/3	16/2
	<i>m</i> <sub>3</sub>	0/0	TO/TO	19.26/199.95	5590/4923	16/64
	<i>m</i> <sub>4</sub>	1/1	11.35 /11.52	11.14 /14.22	3/2	2 /2
	<i>m</i> <sub>5</sub>	1/0	10.98/TO	10.85/13.44	2/1991	2/ 1
Gauss_elim(6/8/10)	<i>o</i>	0/0/0	TO/TO/TO	13.47/15.12/87.45	2756/2055/1662	1/1/1
	<i>m</i> <sub>1</sub>	1/1/1	155.40/TO/TO	14.31/16.99/88.79	121/2131/559	2/2/2
Heat(6/8/10)	<i>o</i>	1/1/1	17.31/17.99/20.51	16.75 /19.27/22.75	2/2/2	1/1/1
	<i>m</i> <sub>1</sub>	1/1/1	17.33/18.21/20.78	17.03 /19.75/23.16	2/2/2	1/1/1
	<i>m</i> <sub>2</sub>	1/1/1	18.35/18.19/20.74	16.36 /19.53/23.07	2/2/2	1/1/1
	<i>m</i> <sub>3</sub>	1/1/1	19.64/20.21/23.08	16.36/19.72/22.95	3/3/3	1/1/1
	<i>m</i> <sub>4</sub>	1/1/1	22.9/24.73/27.78	16.4/19.69/22.90	9/9/9	1/1/1
	<i>m</i> <sub>5</sub>	1/1/1	24.28/28.57/32.67	16.61/19.59/22.42	7/7/7	1/1/1
Mandelbrot(6/8/10)	<i>o</i>	0/0/-1	TO/TO/TO	117.68 / 831.87 /TO	500/491/447	9 / 9 /9
	<i>m</i> <sub>1</sub>	-1/-1/-1	TO/TO/TO	TO/TO/TO	1037/1621/1459	173/227/246
	<i>m</i> <sub>2</sub>	-1/-1/-1	TO/TO/TO	TO/TO/TO	1093/1032/916	178/136/90
	<i>m</i> <sub>3</sub>	1/1/1	10.71/11.17/11.92	10.84/11.68/13.5	2/2/2	2/2/2
Mandelbort*(4/6)	<i>o</i>	0/0	68.09/270.65	12.65/13.21	72/240	2/2
Sorting*(4/6)	<i>o</i>	0/0	TO/TO	19.18/46.19	584/519	1/1
Image_mani(6/8/10)	<i>o</i>	0/0/0	97.69/118.72/141.87	18.68/23.84/27.89	96/96/96	4/4/4
	<i>m</i> <sub>1</sub>	1/1/1	12.92/15.80/15.59	14.15/ 14.53 /16.86	2/2/2	2/2/2
DepSolver(6/8/10)	<i>o</i>	0/0/0	94.17/116.5/148.38	97.19/123.36/151.83	4/4/4	4/4/4
Kfray(6/8/10)	<i>o</i>	0/0/0	TO/TO/TO	51.59/68.25/226.96	1054/981/1146	1/1/1
	<i>m</i> <sub>1</sub>	1/1/1	52.15/53.50/46.83	53.14/69.58/229.97	2/2/2	2/2/2
	<i>m</i> <sub>2</sub>	-1/-1/-1	TO/TO/TO	TO/TO/TO	1603/1583/1374	239/137/21
	<i>m</i> <sub>3</sub>	1/1/1	51.31/43.34/48.33	50.40 /71.15/230.18	2/2/2	2/2/2
Kfray*(4/6)	<i>o</i>	0/0	TO/TO	53.44/282.46	1301/1575	1/1
Clustalw(6/8/10)	<i>o</i>	0/0/0	TO/TO/TO	47.28/79.38/238.37	1234/1105/1162	1/1/1
	<i>m</i> <sub>1</sub>	0/0/0	TO/TO/TO	47.94/80.10/266.16	1365/1127/982	1/1/1
	<i>m</i> <sub>2</sub>	0/0/0	TO/TO/TO	47.71/90.32/266.08	1241/1223/915	1/1/1
	<i>m</i> <sub>3</sub>	1/1/1	895.63/TO/TO	149.71/1083.95/301.99	175/1342/866	5/17/2
	<i>m</i> <sub>4</sub>	0/0/0	TO/TO/TO	47.49/79.94/234.99	1347/1452/993	1/1/1
	<i>m</i> <sub>5</sub>	0/0/0	TO/TO/TO	47.75/80.33/223.77	1353/1289/1153	1/1/1

channels as rendezvous ones; thus, the reduction cannot handle the programs with wildcard receives. MPI-SV leverages model checking to prune redundant paths caused by wildcard receives. We applied MPI-SV on MPI-SPIN's 17 C benchmarks to verify deadlock freedom, and MPI-SV successfully analyzed 15 automatically, indicating the effectiveness. For the remaining two programs, *i.e.*, BlobFlow and Monte, MPI-SV cannot analyze them due to the lack of support for APIs. For the real-world program `gausselim`, MPI-SPIN needs

171s to verify that the model is deadlock-free under 5 processes, while MPI-SV only needs 27s to verify the program automatically. If the number of the processes is 8, MPI-SPIN timed out in 30 minutes, but MPI-SV used 66s to complete verification.

**Temporal properties.** We specify two temporal safety properties  $\varphi_1$  and  $\varphi_2$  for Integrate and Mandelbrot, respectively, where  $\varphi_1$  requires process one cannot receive a message before process two, and  $\varphi_2$  requires process one cannot send a message before process

two. Both  $\varphi_1$  and  $\varphi_2$  can be represented by an LTL formula  $\neg a \mathbf{U} b$ , which requires event  $a$  cannot happen before event  $b$ . We verify Integrate and Mandelbrot under 6 processes. The verification results show that MPI-SV detects the violations of  $\varphi_1$  and  $\varphi_2$ , while pure symbolic execution fails to detect violations.

**Runtime bugs.** MPI-SV can also detect local runtime bugs. During the experiments, MPI-SV finds 5 *unknown* memory access out-of-bound bugs: 4 in DepSolver and 1 in ClustalW.

## 6 RELATED WORK

Dynamic analyses are widely used for analyzing MPI programs. Debugging or testing tools [1, 36, 50, 51, 60, 71, 87] have better feasibility and scalability but depend on specific inputs and running schedules. Dynamic verification techniques, *e.g.*, ISP [83] and DAMPI [84], run MPI programs multiple times to cover the schedules under the same inputs. Böhm et al. [3] propose a state-space reduction framework for the MPI program with *non-deterministic synchronization*. These approaches can detect the bugs depending on specific matchings of wildcard operations, but may still miss inputs related bugs. MPI-SV supports both input and schedule coverages, and a larger scope of verifiable properties. MOPPER [23] encodes the deadlock detection problem under concrete inputs in a SAT equation. Similarly, Huang and Mercer [41] use an SMT formula to reason about a trace of an MPI program for deadlock detection. However, the SMT encoding is specific for the zero-buffer mode. Khanna et al. [47] combines dynamic and symbolic analyses to verify *multi-path* MPI programs. Compared with these path reasoning work in dynamic verification, MPI-SV ensures input space coverage and can verify more properties, *i.e.*, safety and liveness properties in LTL. Besides, MPI-SV employs CSP to enable a more expressive modeling, *e.g.*, supporting conditional completes-before [83] and master-slave pattern [32].

For static methods of analyzing MPI program, MPI-SPIN [73, 74] manually models MPI programs in Promela [38], and verifies the model *w.r.t.* LTL properties [58] by SPIN [37] (*cf.* Section 5.4 for empirical comparison). MPI-SPIN can also verify the consistency between an MPI program and a sequential program, which is not supported by MPI-SV. Bronevetsky [9] proposes parallel control flow graph (pCFG) for MPI programs to capture the interactions between arbitrary processes. But the static analysis using pCFG is hard to be automated. ParTypes [55] uses type checking and deductive verification to verify MPI programs against a protocol. ParTypes’s verification results are sound but incomplete, and independent with the number of processes. ParTypes does not support non-deterministic or non-blocking MPI operations. MPI-Checker [22] is a static analysis tool built on Clang Static Analyzer [15], and only supports intraprocedural analysis of local properties such as double non-blocking and missing wait. Botbol et al. [5] abstract an MPI program to symbolic transducers, and obtain the reachability set based on abstract interpretation [18], which only supports blocking MPI programs and may generate false positives. COMPI [53, 54] uses concolic testing [27, 72] to detect assertion or runtime errors in MPI applications. Ye et al. [89] employs partial symbolic execution [68] to detect MPI usage anomalies. However, these two symbolic execution-based bug detection methods do not support the non-determinism caused by wildcard operations. Luo and Siegel [56]

propose a preliminary deductive method for verifying the numeric properties of MPI programs in an unbounded number of processes. However, this method still needs manually provided verification conditions to prove MPI programs.

MPI-SV is related to the existing work on symbolic execution [48], which has been advanced significantly during the last decade [10, 12, 27, 28, 66, 72, 81, 86, 93]. Many methods have been proposed to prune paths during symbolic execution [4, 19, 34, 43, 92]. The basic idea is to use the techniques such as slicing [44] and interpolation [59] to safely prune the paths. Compared with them, MPI-SV only prunes the paths of the same path constraint but different message matchings or operation interleavings. MPI-SV is also related to the work of automatically extracting session types [63] or behavioral types [52] for Go programs and verifying the extracted type models. These methods extract over-approximation models from Go programs, and hence are sound but incomplete. Compared with them, MPI-SV extracts path-level models for verification. Furthermore, there exists work of combining symbolic execution and model checking [20, 65, 79]. YOGI [65] and Abstraction-driven concolic testing [20] combine dynamic symbolic execution [27, 72] with counterexample-guided abstraction refinement (CEGAR) [16]. MPI-SV focuses on parallel programs, and the verified models are path-level. MPI-SV is also related to the work of unbounded verification for parallel programs [2, 6, 7, 85]. Compared with them, MPI-SV is a bounded verification tool and supports the verification of LTL properties. Besides, MPI-SV is related to the existing work of testing and verification of shared-memory programs [13, 14, 21, 34, 35, 39, 40, 42, 49, 62, 90]. Compared with them, MPI-SV concentrates on message-passing programs. Utilizing the ideas in these work for analyzing MPI programs is interesting and left to the future work.

## 7 CONCLUSION

We have presented MPI-SV for verifying MPI programs with both non-blocking and non-deterministic operations. By synergistically combining symbolic execution and model checking, MPI-SV provides a general framework for verifying MPI programs. We have implemented MPI-SV and extensively evaluated it on real-world MPI programs. The experimental results are promising demonstrate MPI-SV’s effectiveness and efficiency. The future work lies in several directions: (1) enhance MPI-SV to support more MPI operations, (2) investigate the automated performance tuning of MPI programs based on MPI-SV, (3) apply our synergistic framework to other message-passing programs.

## REFERENCES

- [1] Allinea. 2002. Allinea DDT. <http://www.allinea.com/products/ddt/>. (2002).
- [2] Alexander Bakst, Klaus von Gleissenthall, Rami Gökhan Kici, and Ranjit Jhala. 2017. Verifying distributed programs via canonical sequentialization. *PACMPL* 1, OOPSLA (2017), 110:1–110:27.
- [3] Stanislav Böhm, Ondrej Meca, and Petr Jancar. 2016. State-Space Reduction of Non-deterministically Synchronizing Systems Applicable to Deadlock Detection in MPI. In *FM*. 102–118.
- [4] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. 2008. RWset: attacking path explosion in constraint-based test generation. In *TACAS*. 351–366.
- [5] Vincent Botbol, Emmanuel Chailloux, and Tristan Le Gall. 2017. Static Analysis of Communicating Processes Using Symbolic Transducers. In *VMCAI*. 73–90.
- [6] Ahmed Bouajjani and Michael Emmi. 2012. Analysis of recursively parallel programs. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles*

- of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. 203–214.
- [7] Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. 2018. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*. 372–391.
  - [8] Daniel Brand and Pitro Zafropulo. 1983. On communicating finite-state machines. *J. ACM* (1983), 323–342.
  - [9] Greg Bronevetsky. 2009. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*. 1–12.
  - [10] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *EuroSYS*. 183–198.
  - [11] Rajkumar Buyya and others. 1999. High performance cluster computing: architectures and systems. *Prentice Hall* (1999), 999.
  - [12] 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.
  - [13] Sagar Chaki, Edmund M. Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. 2004. Efficient Verification of Sequential and Concurrent C Programs. *Formal Methods in System Design* 25, 2-3 (2004), 129–166.
  - [14] Alessandro Cimatti, Iman Narasamya, and Marco Roveri. 2011. Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. 341–356.
  - [15] Clang. 2016. Clang Static Analyzer. <http://clang-analyzer.lvm.org>. (2016).
  - [16] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *CAV*. 154–169.
  - [17] Edmund M Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT press.
  - [18] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. 238–252.
  - [19] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. 2013. Verifying systems rules using rule-directed symbolic execution. In *ASPLOS*. 329–342.
  - [20] Przemysław Dąca, Ashutosh Gupta, and Thomas A Henzinger. 2016. Abstraction-driven Concolic Testing. In *VMCAI*. 328–347.
  - [21] Brian Demsky and Patrick Lam. 2015. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 20–36.
  - [22] Alexander Droste, Michael Kuhn, and Thomas Ludwig. 2015. MPI-checker: static analysis for MPI. In *LLVM-HPC*. 3:1–3:10.
  - [23] Vojtěch Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2014. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *FM*. 263–278.
  - [24] MPI Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. <http://mpi-forum.org>. (2012).
  - [25] Xianjin Fu, Zhenbang Chen, Yufeng Zhang, Chun Huang, Wei Dong, and Ji Wang. 2015. MPISE: Symbolic Execution of MPI Programs. In *HASE*. 181–188.
  - [26] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhakaran Kambadur, Brian Barrett, Andrew Lumsdaine, and others. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *EuroMPI*. 97–104.
  - [27] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *PLDI*. 213–223.
  - [28] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *NDSS*.
  - [29] 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. [https://science.energy.gov/-/media/ascr/pdf/programdocuments/docs/2017/HPC\\_Correctness\\_Report.pdf](https://science.energy.gov/-/media/ascr/pdf/programdocuments/docs/2017/HPC_Correctness_Report.pdf). (2017).
  - [30] 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.
  - [31] William Gropp. 2002. MPICH2: A new start for MPI implementations. In *EuroMPI*. 7–7.
  - [32] William Gropp, Ewing Lusk, and Anthony Skjellum. 2014. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press.
  - [33] William Gropp, Ewing Lusk, and Rajeev Thakur. 1999. *Using MPI-2: Advanced features of the message-passing interface*. MIT press.
  - [34] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. 2015. Assertion guided symbolic execution of multithreaded programs. In *FSE*. 854–865.
  - [35] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial symbolic execution for detecting concurrency-related cache timing leaks. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 377–388.
  - [36] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R de Supinski, and Matthias S Müller. 2012. MPI runtime error detection with MUST: advances in deadlock detection. In *SC*. 30.
  - [37] Gerard J Holzmann. 1997. The model checker SPIN. *IEEE Transactions on Software Engineering* (1997), 279–295.
  - [38] Gerard J. Holzmann. 2012. Promela manual pages. <http://spinroot.com/spin/Man/promela.html>. (2012).
  - [39] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. 141–152.
  - [40] Shiyu Huang and Jeff Huang. 2016. Maximal causality reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 447–461.
  - [41] Yu Huang and Eric Mercer. 2015. Detecting MPI Zero Buffer Incompatibility by SMT Encoding. In *NFM*. 219–233.
  - [42] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 807–812.
  - [43] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A Navas. 2013. Boosting concolic testing via interpolation. In *FSE*. 48–58.
  - [44] Ranjit Jhala and Rupak Majumdar. 2005. Path slicing. In *PLDI*. 38–47.
  - [45] Ke Jiang and Bengt Jonsson. 2009. Using SPIN to model check concurrent algorithms, using a translation from C to Promela. In *MCC 2009*. 67–69.
  - [46] René Just, Dariosous Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *FSE*. 654–665.
  - [47] Dhriti Khanna, Subodh Sharma, César Rodríguez, and Rahul Purandare. 2018. Dynamic Symbolic Verification of MPI Programs. In *FM*.
  - [48] J.C. King. 1976. Symbolic execution and program testing. *Commun. ACM* (1976), 385–394.
  - [49] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2018. Synchronizing the Asynchronous. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*. 21:1–21:17.
  - [50] Bettina Krammer, Katrin Bidmon, Matthias S Müller, and Michael M Resch. 2004. MARMOT: An MPI analysis and checking tool. *Advances in Parallel Computing* (2004), 493–500.
  - [51] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. 2015. Debugging high-performance computing applications at massive scales. *Commun. ACM* 58, 9 (2015), 72–81.
  - [52] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A static verification framework for message passing in Go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 1137–1148.
  - [53] Hongbo Li, Zizhong Chen, and Rajiv Gupta. 2019. Efficient Concolic Testing of MPI Applications. In *Proceedings of the 28th International Conference on Compiler Construction (CC 2019)*. 193–204.
  - [54] Hongbo Li, Sihuan Li, Zachary Benavides, Zizhong Chen, and Rajiv Gupta. 2018. COMPI: Concolic Testing for MPI Applications. In *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*. 865–874.
  - [55] 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.
  - [56] Ziqing Luo and Stephen F. Siegel. 2018. Towards Deductive Verification of Message-Passing Parallel Programs. In *2nd IEEE/ACM International Workshop on Software Correctness for HPC Applications, CORRECTNESS@SC 2018, Dallas, TX, USA, November 12, 2018*. 59–68.
  - [57] Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. 2017. Verification of MPI programs using CIVL. In *EuroMPI*. 6:1–6:11.
  - [58] Zohar Manna and Amir Pnueli. 1992. *The temporal logic of reactive and concurrent systems - specification*. Springer.
  - [59] Kenneth L. McMillan. 2005. Applications of Craig Interpolants in Model Checking. In *TACAS*. 1–12.
  - [60] Subrata Mitra, Ignacio Laguna, Dong H. Ahn, Saurabh Bagchi, Martin Schulz, and Todd Gamblin. 2014. Accurate application progress analysis for large-scale

- parallel debugging. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 193–203.
- [61] Matthias Müller, Bronis de Supinski, Ganesh Gopalakrishnan, Tobias Hilbrich, and David Lecomber. 2011. Dealing with MPI bugs at scale: Best practices, automatic detection, debugging, and formal verification. [http://sc11.supercomputing.org/schedule/event\\_detail.php?evind=tut131](http://sc11.supercomputing.org/schedule/event_detail.php?evind=tut131), (2011).
- [62] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Pira-manayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 267–280.
- [63] Nicholas Ng and Nobuko Yoshida. 2016. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 174–184.
- [64] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. Springer.
- [65] Aditya V Nori, Sriram K Rajamani, SaiDeep Tetali, and Aditya V Thakur. 2009. The YOGI Project: Software property checking via static analysis and testing. In *TACAS*. 178–181.
- [66] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. 2008. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. 15–26.
- [67] Wojciech Penczek, Maciej Szreter, Rob Gerth, and Ruurd Kuiper. 2000. Improving Partial Order Reductions for Universal Branching Time Properties. *Fundam. Inform.* (2000), 245–267.
- [68] David A. Ramos and Dawson R. Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *SEC. USENIX Association*, 49–64.
- [69] Juan A. Rico-Gallego and Juan Carlos Diaz Martin. 2011. Performance Evaluation of Thread-Based MPI in Shared Memory. In *EuroMPI*. 337–338.
- [70] Bill Roscoe. 2005. *The theory and practice of concurrency*. Prentice-Hall.
- [71] 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.
- [72] Koushik Sen, Darko Marinov, and Gul Agha. 2005. 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, 2005, Lisbon, Portugal, September 5-9, 2005*. 263–272.
- [73] Stephen F. Siegel. Model Checking Nonblocking MPI Programs. In *VMCAI*.
- [74] Stephen F. Siegel. 2007. Verifying Parallel Programs with MPI-Spin. In *PVM/MPI*. 13–14.
- [75] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: the concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. 61:1–61:12.
- [76] Stephen F Siegel and Timothy K Zirkel. 2011. FEVS: A functional equivalence verification suite for high-performance scientific computing. *Mathematics in Computer Science* (2011), 427–435.
- [77] Stephen F. Siegel and Timothy K. Zirkel. 2011. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science* (2011), 395–426.
- [78] Marc Snir. 1998. *MPI—the Complete Reference: The MPI core*. Vol. 1. MIT press.
- [79] Ting Su, Zhoulai Fu, Geguang Pu, Jifeng He, and Zhendong Su. 2015. Combining symbolic execution and model checking for data flow testing. In *ICSE*. 654–665.
- [80] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. 2009. PAT: Towards flexible verification under fairness. In *CAV*. 709–714.
- [81] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *TAP*. 134–153.
- [82] Sarvani Vakkalanka. 2010. *Efficient dynamic verification algorithms for MPI applications*. Ph.D. Dissertation. The University of Utah.
- [83] 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.
- [84] Anh Vo, Sriram Ananthakrishnan, Ganesh Gopalakrishnan, Bronis R De Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A scalable and distributed dynamic formal verifier for MPI programs. In *SC*. 1–10.
- [85] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL* 3, POPL (2019), 59:1–59:30.
- [86] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 291–302.
- [87] Rogue Wave. 2009. TotalView Software. <http://www.roguewave.com/products/totalview/>. (2009).
- [88] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey Voelker. 2009. MPIWiz: subgroup reproducible replay of MPI applications. *ACM Sigplan Notices* (2009), 251–260.
- [89] Fangke Ye, Jisheng Zhao, and Vivek Sarkar. 2018. Detecting MPI usage anomalies via partial program symbolic execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. 63:1–63:5.
- [90] Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. 2018. YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement - (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*. 422–426.
- [91] Hengbiao Yu, Zhenbang Chen, Xianjin Fu, Ji Wang, Zhendong Su, Jun Sun, Chun Huang, and Wei Dong. 2020. Combining Symbolic Execution and Model Checking to Verify MPI Programs. *CoRR* abs/1803.06300 (2020). arXiv:1803.06300 <http://arxiv.org/abs/1803.06300>
- [92] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic verification of regular properties. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 871–881.
- [93] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular property guided dynamic symbolic execution. In *ICSE. IEEE Press*, 643–653.