Symbolic Verification of Message Passing Interface Programs

Hengbiao Yu∗, Zhenbang Chen†, Xianjin Fu¹², Ji Wang¹², Zhendong Su³, Jun Sun⁴, Chun Huang⁴, Wei Dong¹
¹College of Computer, National University of Defense Technology, Changsha, China
²State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China
³Department of Computer Science, ETH Zurich, Switzerland
⁴School 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 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

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.

ACM Reference Format:


∗The first two authors contributed equally to this work and are co-first authors. Zhenbang Chen and Ji Wang are the corresponding authors.
†MPI-SV is available at 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.

ICSE ’20, May 23-29, 2018, Seoul, South Korea
© 2020 Association for Computing Machinery.
ACM ISBN xxxx-xxxx-xxxx-xxxx/xx/xx...
https://doi.org/XX.XXXX/XXXXXXX.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 [53]. 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 [14, 20, 25, 24] 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 [27, 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 [55] 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) [55]. If the extracted model from a path satisfies the property \( \phi \), the equivalent paths can be safely pruned; otherwise, if the model checker reports a counterexample, a violation of \( \phi \) 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 [90]. We have used MPI-SV to analyze 12 real-world MPI programs, totaling 47K lines of code (LOC) (three are implemented tasks, MPI-SV achieves, on average, 19x speedups on verifying tasks, i.e., deadlock freedom verification tasks, where 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 program 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 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 \( T \) be a set of types, \( N \) a set of names, and \( E \) a set of expressions. For simplifying the discussion, we define a core language for MPI programs in Figure 1, where \( T \in T, r \in N, \) and \( e \in E \). An MPI program \( MP \) 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 \( \text{var } r : T \) declares a variable \( r \) with type \( T \). The statement \( \text{r } := e \) assigns the value of expression \( e \) to variable \( r \). 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 \( e \) indicates the destination process’s identifier. These message passing operations can be blocking or non-blocking.

First, we introduce blocking operations:

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

A \( \text{Recv}(*,r) \) operation, called wildcard receive, may receive a message from different processes under different runs, resulting in non-determinism. The blocking of a \( \text{Send}(e) \) 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 \( \text{Send}(e) \) 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.

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

\[
\begin{align*}
\text{Comm} & : = \text{Send}(e) | \text{Send}(e) | \text{Recv}(e) | \text{Recv}(*,r) | \text{Barrier} | \\
& \text{ISend}(e,r) | \text{IREcv}(e,r) | \text{IREcv}(*,r) | \text{Wait}(r)
\end{align*}
\]

Figure 1: Syntax of a core MPI language.
Symbolic Verification of Message Passing Interface Programs

<table>
<thead>
<tr>
<th>P₀</th>
<th>P₁</th>
<th>P₂</th>
<th>P₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>if (x != 'a')</td>
<td>Recv(0)</td>
<td>Send(1)</td>
</tr>
<tr>
<td></td>
<td>else</td>
<td>IRecv(*,req);</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Recv(3)</td>
<td></td>
</tr>
</tbody>
</table>

**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 [21].

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 P₀, P₂ and P₃ only send a message to P₁ and then terminate. For process P₁, if input x is not equal to 'a', P₁ receives a message from P₀ in a blocking manner; otherwise, P₁ uses a non-blocking wildcard receive to receive a message. Then, P₁ receives a message from P₃. When x is 'a' and IRecv(*,req) receives the message from P₃, a deadlock occurs, i.e., P₁ 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 [43]. 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 [53] 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 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 Γ, 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 Γ w.r.t. the property. If the model checker reports a counterexample, a violation is found; otherwise, if Γ 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 P₁ receives a message from P₃. MPI-SV first symbolically executes P₀, which only sends a message to P₁. The Send(1) operation returns immediately with the assumption of infinite system buffers. Hence, P₀ terminates, and the operation Send(1) is recorded. Then, MPI-SV executes P₁ and explores both branches of the conditional statement as follows.

1. **True branch** (x ≠ 'a'). In this case, P₁ blocks at Recv(0). MPI-SV records the receive operation for P₁ and starts executing P₂. Like P₀, P₂ executes operation Send(1) and terminates, after which P₃ is selected and behaves the same as P₂. After P₃ terminates, the global execution blocks, i.e., P₁ blocks and all the other processes

**Figure 3:** The framework of MPI-SV.

**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 `Recv(θ)` in `P1` should be matched with the `Send(1)` in `P0`. After executing the send and receive operations, MPI-SV selects `P1` to execute, because `P0` terminates. Then, `P1` blocks at `Recv(3)`. Same as earlier, the global execution blocks and operation matching needs to be done. `Recv(3)` is matched with the `Send(1)` in `P3`. After executing the `Recv(3)` and `Send(1)` operations, all the processes terminate successfully. Path `P1` in Figure 3 is explored.

(2) False branch (x = 'a'). The execution of `P1` proceeds until reaching the blocking receive `Recv(3)`. Additionally, the two issued receive operations, i.e., `IRrecv(*, req)` and `Recv(3)`, are recorded. Similar to the true branch, when each process blocks or terminates, we handle operation matching. Here `P0`, `P2` and `P3` terminate, and `P1` blocks at `Recv(3)`. `IRrecv(*, req)` should be matched first because of the non-overlapped policy in the MPI standard [24]. There are three `Send` operation candidates from `P0`, `P2` and `P3`, respectively. MPI-SV forks a state for each candidate. Suppose MPI-SV first explores the state where `IRrecv(*, req)` is matched with `P0`'s `Send(1)`. After matching and executing `P1`'s `Recv(3)` and `P0`'s `Send(1)`, the path terminates successfully, which generates path `p2` in Figure 3.

**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., `p4` in Figure 3), where `IRrecv(*, req)` is matched with `P3`'s `Send(1)` and `P1` blocks because `Recv(3)` has no matched operation. MPI-SV generates a CSP model `Γ` based on the deadlock-free path `p2` where `P1`'s `IRrecv(*, req)` is matched with `P3`'s `Send(1)`. Each MPI process is modeled as a CSP process, and all the CSP processes are composed in parallel to form `Γ`. Notably, in `Γ`, 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 [23], are also modeled. A CSP model checker then verifies deadlock freedom for `Γ`. The model checker reports a counterexample where `IRrecv(*, req)` is matched with the `Send(1)` in `P3`. MPI-SV only explores two paths for detecting the deadlock and avoids the exploration of `p3` and `p4` (indicated by dashed lines).

**Pruning.** Because the CSP modeling is precise (cf. Section 2), 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 `P1` is a `Recv(*, req)` operation. Then, the program is deadlock free. The true branch (x = 'a') has 2 paths, because the last wildcard receive in `P1` has two matchings (i.e., `P2`'s `send` and `P3`'s `send`, and `P0`'s `send` has been matched by `P1`'s `Recv(θ)`). The false branch (x = 'a') has 6 paths because the first wildcard receive has 3 matchings (send operations from `P0`, `P2` and `P3`) 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 `Send(1)` operation in `P0` should be completed before the `Send(1)` operation in `P2`. Actually, the send operation in `P2` can be completed before the send operation in `P0`, 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 `MP = {Proc_i | 0 ≤ i ≤ n}`, a state `S` in `MP`'s symbolic execution is composed by the states of processes, i.e., `{s_0, ..., s_n}`, and each MPI process’s state is a 6-tuple `(M, Stat, PC, F, B, R)`, where `M` maps each variable to a concrete value or a symbolic value, `Stat` is the next program statement to execute, `PC` is the process’s path constraint [24], `F` is the flag of process status belonging to `{active, blocked, terminated}`, `B` and `R` are infinite buffers for storing the issued MPI operations not yet matched and the matched MPI operations, respectively. We use `s_i ∈ S` to denote that `s_i` is a process state in the global state `S`. An element `elem` of `s_i` can be accessed by `s_i.elem`, e.g., `s_i.F` is the `i`th process’s status flag. In principle, a statement execution in any process advances the global state, making `MP`’s state space exponential to the number of processes. We use variable `Seq_i` defined in `M` to record the sequence of the issued MPI operations in `Proc_i`, and `Seq(S)` to denote the set `{Seq | 0 ≤ i ≤ n}` of global state `S`. Global state `S`’s path condition (denoted by `S`, `PC`) is the conjunction of the path conditions of `S`’s processes, i.e., `\bigwedge_{i \in S} s_i.PC_i ∈ S`.

Algorithm 3 shows the details of MPI-SV. We use `worklist` to store the global states to be explored. Initially, `worklist` only contains `Sinit`, composed of the initial states of all the processes, and each process’s status is active. At Line 9, 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 `Proc_i` to execute. Next, Execute (cf. Algorithm 2) symbolically executes the statement `Stat_i` in `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 `Stat_i`, if all the processes in the current global state `S` terminate, i.e., a violation-free path terminates, we use Algorithm 4 to generate a CSP model `Γ` from the current state (Line 3). Then, we use a CSP model checker to verify `Γ` w.r.t. `Γ`. If `Γ` satisfies `φ` (denoted by `Γ ⊨ φ`), 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`’s path condition; otherwise, if the model checker gives a counterexample, we report the violation and exit (Line 14).
Algorithm 1: Symbolic Verification Framework

Data: \( M_P \) is \( \{ Pro_{ci} \ | \ 0 \leq i \leq n \} \), \( \varphi \) is a property, and Sym is a set of symbolic variables

1 begin
2 worklist \( \leftarrow \{ S_{init} \} \)
3 while worklist \( \neq \emptyset \) do
4 \( S_c \leftarrow \text{Select(worklist)} \)
5 \( (M_i, Stat_i, PC_i, F_i, B_i, R_i) \leftarrow \text{Scheduler}(S_c) \)
6 Execute\( (S_c, Pro_{ci}, Stat_i, Sym, \text{worklist}) \)
7 if \( \forall S_j \in S_c, S_j. F = \text{terminated} \) then
8 \( \Gamma \leftarrow \text{GenerateCSP}(S_c) \)
9 ModelCheck\( (\Gamma, \varphi) \)
10 if \( \Gamma \models \varphi \) then
11 worklist \( \leftarrow \text{worklist}\backslash \{ S_p \in \text{worklist} | S_p, PC \Rightarrow S_c, PC \} \)
12 end
13 else if \( \Gamma \nvdash \varphi \) then
14 reportViolation and Exit
15 end
16 end
17 end
18 end

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 [18] (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--9). We also need to update buffer \( B \) after issuing an MPI operation (Lines 5--9). Then, if \( Stat_i \) is a non-blocking operation, the execution returns immediately; otherwise, we block \( Pro_{ci} \) (Line 10 excepting the \( \text{Wait} \) of an \( ISend \) operation). When reaching GlobalBlocking (Line 11), i.e., every process is terminated or blocked, we Match (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 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 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 Fire\( (S_c, pair_r) \) (Line 5). If there is no matching for non-wildcard operations, we use match\( W \) to match the wildcard operations (Line 5). For each possible matching of a wildcard receive, we fork a new state (denoted by fork\( (S_c, pair_r) \) at Line 10 to analyze each matching case. If no operations can be matched, but there exist blocked processes, a

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., \( Pro_{ci} \) 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.
4.1 CSP Subset

Let \( \Sigma \) be a \textit{finite} set of events, \( C \) 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, \( a \in \Sigma, c \in C, X \subseteq \Sigma \) and \( x \in X \).

\[
P := a \mid P ; P \mid P + P \mid P \parallel P \mid P \mid c?x \rightarrow P \mid c!x \rightarrow P \mid \text{skip}
\]

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 (\( \mid \)), external choice (\( \parallel \)) and parallel composition with synchronization (\( \parallel \)). \( P \parallel Q \) performs as \( P \) or \( Q \), and the choice is made by the environment. Let \( PS \) be a finite set of processes, \( \square PS \) denotes the external choice of all the processes in \( PS \), \( P \parallel Q \) 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 \text{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 \( \text{Proc}_i \), we

\begin{algorithm}
\begin{footnotesize}
\begin{algorithmic}
\State \textbf{GenerateCSP(S)}
\State \textbf{Data:} A terminated global state \( S \), and\n\State \( \text{Seq}_i = \{ \text{Seq}_i \mid 0 \leq i \leq n \} \)
\State \textbf{begin}
\State \hspace{1em} \( PS \leftarrow \emptyset \)
\State \hspace{1em} \textbf{for} \( i \leftarrow 0 \ldots n \) \textbf{do}
\State \hspace{2em} \( P_i \leftarrow \text{skip} \)
\State \hspace{2em} \( \text{Req} \leftarrow \{ r \mid \text{IRcv}(\ast, r) \in \text{Seq}_i \lor \text{IRcv}(1, r) \in \text{Seq}_i \} \)
\State \hspace{2em} \textbf{for} \( j \leftarrow \text{length(\text{Seq}_i)} - 1 \ldots 0 \) \textbf{do}
\State \hspace{3em} \textbf{switch} \( \text{op}_j \) \textbf{do}
\State \hspace{4em} \textbf{case} \text{Send}(i) \textbf{ or } \text{Send}(i, r) \textbf{ do}
\State \hspace{5em} \( c_2 \leftarrow \text{Chan}(\text{op}_j) \) \hspace{1em} // \( c_2 \)’s size is 0
\State \hspace{5em} \( P_i \leftarrow c_2!x \rightarrow P_i \)
\State \hspace{3em} \textbf{end}
\State \hspace{2em} \textbf{case} \text{Recv}(i) \textbf{ or } \text{Recv}(\ast) \textbf{ do}
\State \hspace{3em} \( C \leftarrow \text{StaticMatchedChannel}(\text{op}_j, S) \)
\State \hspace{3em} \( Q \leftarrow \text{Refine}(\square \{ c?x \rightarrow \text{skip} \mid c \in C \}, S) \)
\State \hspace{3em} \( P_i \leftarrow Q \parallel P_i \)
\State \hspace{3em} \textbf{end}
\State \hspace{2em} \textbf{case} \text{IRcv}(\ast, r) \textbf{ or } \text{IRcv}(1, r) \textbf{ do}
\State \hspace{3em} \( C \leftarrow \text{StaticMatchedChannel}(\text{op}_j, S) \)
\State \hspace{3em} \( Q \leftarrow \text{Refine}(\square \{ c?x \rightarrow \text{skip} \mid c \in C \}, S) \)
\State \hspace{3em} \( e_w \leftarrow \text{WaitEvent}(\text{op}_j) \) \hspace{1em} // \( \text{op}_j \)’s wait event
\State \hspace{3em} \( P_i \leftarrow (Q \parallel e_w) \parallel P_i \)
\State \hspace{3em} \textbf{end}
\State \hspace{2em} \textbf{end}
\State \hspace{1em} \textbf{case} \text{Wait}(r) \textbf{ and } r \in \text{Req} \textbf{ do}
\State \hspace{2em} \( e_w \leftarrow \text{GenerateEvent}(\text{op}_j) \)
\State \hspace{2em} \( P_i \leftarrow e_w \parallel P_i \)
\State \hspace{2em} \textbf{end}
\State \hspace{1em} \textbf{end}
\State \hspace{1em} \text{PS} \leftarrow PS \parallel \{ P_i \}
\State \hspace{1em} \textbf{end}
\State \hspace{1em} \textbf{return} \( P \)
\end{algorithmic}
\end{footnotesize}
\end{algorithm}

Theorem 1: Correctness. Blocking-driven symbolic execution is an instance of model checking with POR. We have proved the symbolic execution method is correct for \textit{reachability properties} \cite{59}. Due to the space limit, the proof can be referred to \cite{91}.

The single event process \( a \) performs the event \( a \) and terminates. There are three operators: sequential composition (\( \mid \)), external choice (\( \parallel \)) and parallel composition with synchronization (\( \parallel \)). \( P \parallel Q \) performs as \( P \) or \( Q \), and the choice is made by the environment. Let \( PS \) be a finite set of processes, \( \square PS \) denotes the external choice of all the processes in \( PS \), \( P \parallel Q \) 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 \text{skip} terminates immediately.

Theorem 1: Correctness. Blocking-driven symbolic execution is an instance of model checking with POR. We have proved the symbolic execution method is correct for \textit{reachability properties} \cite{59}. Due to the space limit, the proof can be referred to \cite{91}.

Figure 5: An example of operation matching.
Symbolic Verification of Message Passing Interface Programs

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 \( \text{Chan}(op) \). We use a zero-sized channel to model Send operation (Line 10), because Send blocks until the message is received. In contrast, considering a Send or 1Send 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 17-18). 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 23).

StaticMatchedChannel \((op, \mathcal{S})\) (Lines 20-25) returns the set of the channels written by the possibly matched send operations of the receive operation \( op \). We scan \( Seq(S) \) to obtain the possibly matched send operations of \( op \). Given a receive operation \( \text{recv} \) in process \( \text{Proc}_i \), SMO \((\mathcal{S}), \mathcal{S}\text{recv} \) calculated as follows denotes the set of the matched send operations of \( \text{recv} \).

- If \( \text{recv} \) is \( \text{Recv}(j) \) or \( \text{IRecv}(j, r) \), SMO \((\mathcal{S}), \mathcal{S}\text{recv} \) contains \( \text{Proc}_j \)’s send operations with \( \text{Proc}_j \) as the destination process.
- If \( \text{recv} \) is \( \text{Recv}(+), \text{IRecv}(+, r) \), SMO \((\mathcal{S}), \mathcal{S}\text{recv} \) contains any process’s send operations of \( \text{Proc}_j \) as the destination process.

SMO \((op, \mathcal{S})\) 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 \( \text{Proc}_0 \) be \( \text{Send}(1) \text{Barrier.Send}(1) \), and \( \text{Proc}_1 \) be \( \text{Recv}(+) \text{Barrier} \). SMO will add the two send operations in \( \text{Proc}_0 \) to the matching set of the \( \text{Recv}(+) \) in \( \text{Proc}_1 \). Since \( \text{Recv}(+) \) must complete before \( \text{Barrier} \), we can remove the second send operation in \( \text{Proc}_0 \). Such optimization reduces the complexity of the CSP model. For brevity, we use SMO \((op, \mathcal{S})\) to denote the optimized matching set. Then, StaticMatchedChannel \((op, \mathcal{S})\) is \( \{ \text{Chan}(op) \mid op \in \text{SMO}(op, \mathcal{S}) \} \).

To satisfy the MPI requirements, Refine \((P, \mathcal{S})\) (Lines 21-26) refines the models of receive operations by imposing the completeness-before requirements \([33]\) 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 completeness-before pattern \([33]\). 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 1Send 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 \( \epsilon_w \) for each Wait operation (Line 31). Then, \( \epsilon_w \) is produced after the corresponding non-blocking operation is completed (Line 29). The synchronization on \( \epsilon_w \) ensures that a Wait operation blocks until the corresponding non-blocking operation is completed.

We use the example in Figure 3 for a demonstration. After exploring a violation-free path, the recorded operation sequences are \( \text{Seq}_{\text{rec}}=(\text{ISend}(1, r_{\text{req}}), \text{Barrier.Wait}(r_{\text{req}})), \text{Seq}_{\text{rec}}=(\text{IRrecv}(+, r_{\text{req}}), \text{Barrier.Wait}(r_{\text{req}})), \text{Seq}_{\text{rec}}=(\text{Barrier.ISend}(1, r_{\text{req}}), \text{Wait}(r_{\text{req}})) \). We first scan \( \text{Seq}_{\text{rec}} \) in reverse. Note that we don’t model \( \text{Wait}(r_{\text{req}}) \), because it corresponds to \( \text{IRecv} \). We create a synchronization event \( \text{B} \) for modeling Barrier (Lines 16-17). For the ISend(1, \( r_{\text{req}} \) ), we model it by writing an element \( a \) to a one-sized channel \( \text{chan}_1 \), and use prefix operation to compose its model with \( \text{B} \) (Lines 12-14). In this way, we generate CSP process \( \text{chan}_1 \rightarrow \text{B} \vdash \text{skip} \) (denoted by CP\(_0\)) for \( \text{Proc}_0 \). Similarly, we model CP\(_2\) by \( \text{B} \vdash \text{chan}_2 \rightarrow \text{skip} \) (denoted by CP\(_2\)) where \( \text{chan}_2 \) is also a one-sized channel and \( \text{B} \) is a channel element. For \( \text{Proc}_1 \), we generate a single event \( \epsilon_w \) to model \( \text{Wait}(r_{\text{req}}) \), because it corresponds to \( \text{IRrecv} \) (Lines 30-32). For \( \text{IRrecv}(+, r_{\text{req}}) \), we first compute the matched channels using SMO (Line 25), and StaticMatchedChannel \((op, \mathcal{S})\) contains both \( \text{chan}_1 \) and \( \text{chan}_2 \). Then, we generate the following CSP process

\[
\left( (\text{chan}_1 \rightarrow \text{skip} \circ \text{chan}_2 \rightarrow \text{skip}) ; \varepsilon_w \right) \parallel \left( \text{B} ; \varepsilon_w ; \text{skip} \right) \langle \epsilon_w \rangle
\]

(denoted by CP\(_1\)) for \( \text{Proc}_1 \). Finally, we compose the CSP processes using the parallel operator to form the CSP model (Line 35, i.e., CP\(_0\) \parallel CP\(_1\) \parallel CP\(_2\), (8)) (8)

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 CP\(_S\). Here, soundness means that CP\(_S\) 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 CP\(_S\) represents a real behavior that can be derived from \( S \) by changing the matchings or interleavings of the communications.

Since we compute SMO \((op, \mathcal{S})\) by statically matching the arguments of the recorded operations, SMO \((op, \mathcal{S})\) 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\(_{\text{static}}\) and CSP\(_{\text{ideal}}\) to denote the generated models
using 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 CSP_{\text{static}}’s
consistency to the MPI semantics, which imply the soundness and
completeness of our CSP modeling method. Let \( T(P) \) denote the
trace set [70] of CSP process \( P \), and \( T(F) \) denote the failure set of
CSP process \( P \). Each element in \( F(P) \) is \((s, X)\), where \( s \in T(P) \) is a trace, and \( X \) is
the set of events \( P \) refuses to perform after \( s \).

**Theorem 4.1.** \( F(CSP_{\text{static}}) = F(CSP_{\text{ideal}}) \).

**Proof.** We only give the skeleton of the proof. We first prove
\( T(CSP_{\text{static}}) \equiv T(CSP_{\text{ideal}}) \)

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

Suppose there is a trace \( t = \langle e_1, ..., e_n \rangle \) such that \( t \in T(CSP_{\text{static}}) \)
\( \text{ but not } t \in T(CSP_{\text{ideal}}) \). The only difference between
\( CSP_{\text{static}} \) and \( CSP_{\text{ideal}} \) is that \( CSP_{\text{static}} \) 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 \( e_k = \lambda x \cdot c \), where \( 1 \leq k \leq n \). Therefore, \( c \)
cannot be read in \( CSP_{\text{ideal}} \) when the matching of the corresponding
receive operation starts, but \( c \) is not empty at \( e_k \) in \( CSP_{\text{static}} \).

Despite of the size of \( c \), there must exist a write operation \( \lambda y \cdot e \)
in \( \langle e_1, ..., e_{k-1} \rangle \). Because \( \langle e_1, ..., e_{k-1} \rangle \) is also a valid trace in
\( CSP_{\text{ideal}} \), it means \( c \) is not empty in \( CSP_{\text{ideal}} \) at \( e_k \), which contradicts
with the assumption that \( c \) cannot be read in \( CSP_{\text{ideal}} \). Hence,
\( T(CSP_{\text{static}}) \subseteq T(CSP_{\text{ideal}}) \) holds. \( \square \)

**Theorem 4.2.** \( CSP_{\text{static}} \) is consistent with the MPI semantics.

The proof’s main idea is to prove that \( CSP_{\text{ideal}} \) 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 \( CSP_{\text{static}} \) 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 Cloudy’s
support for multi-threaded programs. We use a multi-threaded li-
brary for MPI, called AzequaMPI [50], 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 sym-
bolic execution. We use Clang to compile an MPI program to LLVM
bytecode, which is then linked with the pre-compiled MPI library
AzequaMPI. The symbolic execution module is in charge of path
exploration and property checking. The third module utilizes the

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

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 effec-
tively? 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 pro-
grams are real-world open source MPI programs. DTG is a testing
program from [62]. Matmat, Integrate, and Diffusion2d come from
the FEVS benchmark suite [70]. 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 [83]. Heat is a parallel solver for heat equa-
tion 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 \( \text{if } (x > a) \{ \text{Recv}(i) \} \text{ else } \text{Recv}(r) \); (2) replace \( \text{Recv}(r) \)
with \( \text{if } (x > a) \{ \text{Recv}(r) \} \text{ else } \text{Recv}(s) \). 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 \( \text{IRcv}(i, s) \) and \( \text{IRcv}(r, r) \)
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 [33], 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_i represents a mutant. A task comprises a program and the number of running processes. We label the programs using master-slave pattern with superscript “m”. 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 70 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 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.

<table>
<thead>
<tr>
<th>Program (#Procs)</th>
<th>T</th>
<th>Deadlock</th>
<th>Time(s)</th>
<th>#Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Symbolic execution</td>
<td>Our approach</td>
</tr>
<tr>
<td>DTG(5)</td>
<td>o</td>
<td>0</td>
<td>10.12</td>
<td>9.02</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0</td>
<td>13.69</td>
<td>9.50</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>1</td>
<td>10.02</td>
<td>8.93</td>
</tr>
<tr>
<td></td>
<td>m₃</td>
<td>1</td>
<td>10.21</td>
<td>9.49</td>
</tr>
<tr>
<td></td>
<td>m₄</td>
<td>1</td>
<td>10.08</td>
<td>9.19</td>
</tr>
<tr>
<td></td>
<td>m₅</td>
<td>1</td>
<td>9.04</td>
<td>9.29</td>
</tr>
<tr>
<td>Matlab(4)</td>
<td>o</td>
<td>0</td>
<td>36.94</td>
<td>10.43</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0/0/0</td>
<td>78.17/to/to</td>
<td>8.87/10.45/44.00</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>1/1/1</td>
<td>9.35/9.83/9.94</td>
<td>9.39/10.76/44.09</td>
</tr>
<tr>
<td>Integrate(6/8/10)</td>
<td>o</td>
<td>0</td>
<td>24.18/123.55</td>
<td>9.39/32/83</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0</td>
<td>106.86/to</td>
<td>9.84/13.39</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>0/1</td>
<td>110.25/to/to</td>
<td>10.18/13.81</td>
</tr>
<tr>
<td></td>
<td>m₃</td>
<td>0/0</td>
<td>3236.02/to/to</td>
<td>17.05/14.38</td>
</tr>
<tr>
<td></td>
<td>m₄</td>
<td>0/0</td>
<td>to/to</td>
<td>19.26/199.99</td>
</tr>
<tr>
<td></td>
<td>m₅</td>
<td>1/1</td>
<td>11.35/11.92</td>
<td>11.14/12.22</td>
</tr>
<tr>
<td>Heat(6/8/10)</td>
<td>o</td>
<td>0</td>
<td>155.40/to/to</td>
<td>14.31/16.99/88.79</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0/0/1</td>
<td>17.31/17.99/20.51</td>
<td>16.75/19.27/22.75</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>0/1/1</td>
<td>17.35/18.21/20.78</td>
<td>17.03/19.75/23.16</td>
</tr>
<tr>
<td></td>
<td>m₃</td>
<td>0/1/1</td>
<td>18.35/18.19/20.74</td>
<td>16.36/19.53/23.07</td>
</tr>
<tr>
<td></td>
<td>m₄</td>
<td>0/1/1</td>
<td>19.64/20.21/23.08</td>
<td>16.36/19.72/22.95</td>
</tr>
<tr>
<td></td>
<td>m₅</td>
<td>0/1/1</td>
<td>22.9/24.73/27.78</td>
<td>16.4/19.69/22.90</td>
</tr>
<tr>
<td></td>
<td>o</td>
<td>0/0/1</td>
<td>117.68/to/to</td>
<td>831.87/to/to</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>-1/-1/-1</td>
<td>to/to/to</td>
<td>to/to/to</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>-1/-1/-1</td>
<td>to/to/to</td>
<td>to/to/to</td>
</tr>
<tr>
<td></td>
<td>m₃</td>
<td>1/1/1</td>
<td>10.71/11.17/11.92</td>
<td>10.84/11.68/13.5</td>
</tr>
<tr>
<td>Mandelbrot(6/8/10)</td>
<td>o</td>
<td>0</td>
<td>68.09/270.65</td>
<td>12.65/13.21</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0</td>
<td>19.48/16.19</td>
<td>584/519</td>
</tr>
<tr>
<td></td>
<td>Image_man(6/8/10)</td>
<td>o</td>
<td>0/0/0</td>
<td>97.69/118.72/141.87</td>
</tr>
<tr>
<td></td>
<td>DepSolver(6/8/10)</td>
<td>o</td>
<td>0/0/0</td>
<td>94.17/116.5/148.38</td>
</tr>
<tr>
<td></td>
<td>Kfray(6/8/10)</td>
<td>o</td>
<td>0/0/0</td>
<td>50.59/68.25/226.96</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0/0/0</td>
<td>51.52/53.50/46.83</td>
<td>53.14/69.58/229.97</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>-1/-1/-1</td>
<td>to/to/to</td>
<td>to/to/to</td>
</tr>
<tr>
<td></td>
<td>m₃</td>
<td>1/1/1</td>
<td>51.31/43.34/48.33</td>
<td>50.40/71.15/230.18</td>
</tr>
<tr>
<td>Kfray*(4/6)</td>
<td>o</td>
<td>0/0/0</td>
<td>to/to/to</td>
<td>53.44/282.46</td>
</tr>
<tr>
<td></td>
<td>m₁</td>
<td>0/0/0</td>
<td>47.28/79.38/238.37</td>
<td>1254/1105/1162</td>
</tr>
<tr>
<td></td>
<td>m₂</td>
<td>0/0/0</td>
<td>47.94/80.10/266.16</td>
<td>1365/1127/982</td>
</tr>
<tr>
<td></td>
<td>m₃</td>
<td>0/0/0</td>
<td>47.71/90.32/266.08</td>
<td>1241/1223/915</td>
</tr>
<tr>
<td></td>
<td>m₄</td>
<td>1/1/1</td>
<td>895.83/to/to</td>
<td>149.31/1083.95/301.99</td>
</tr>
<tr>
<td></td>
<td>m₅</td>
<td>0/0/0</td>
<td>to/to/to</td>
<td>47.49/79.94/234.99</td>
</tr>
<tr>
<td></td>
<td>m₆</td>
<td>0/0/0</td>
<td>to/to/to</td>
<td>47.75/80.33/223.77</td>
</tr>
</tbody>
</table>

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 gauss_e1m, MPI-SV 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 6s 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 $\phi_1$ and $\phi_2$ can be represented by an LTL formula $1\land U b$, 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 $\phi_1$ and $\phi_2$, 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][5][6][51][60][71][87] have better feasibility and scalability but depend on specific inputs and running schedules. Dynamic verification techniques, e.g., ISP [53] and DAMPI [54], run MPI programs multiple times to cover the schedules under the same inputs. Böhm et al. [1] propose a state-space expansion 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 many 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 Mercier [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-befores [53] 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 [55] by SPIN [57] (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 [29] 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 [53] 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. [15] 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 concilic testing [27][72] to detect assertion or runtime errors in MPI applications. Ye et al. [89] employs partial symbolic execution [69] 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 [38], which has been advanced significantly during the last decade [10][12][27][28][60][72][81][86][93]. Many methods have been proposed to prune paths during symbolic execution [11][19][54][43][92]. The basic idea is to use the techniques such as slicing [43] and interpolation [52] 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 concilic 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 [12][72][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][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

Symbolic Verification of Message Passing Interface Programs


