Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs

Vojtěch Forejt, Department of Computer Science, University of Oxford
Saurabh Joshi[1], IIT Hyderabad
Daniel Kroening, Department of Computer Science, University of Oxford
Ganesh Narayanaswamy, Department of Computer Science, University of Oxford
Subodh Sharma, IIT Delhi

The Message Passing Interface (MPI) is the standard API for parallelization in high-performance and scientific computing. Communication deadlocks are a frequent problem in MPI programs, and this paper addresses the problem of discovering such deadlocks. We begin by showing that if an MPI program is *single-path*, the problem of discovering communication deadlocks is NP-complete. We then present a novel propositional encoding scheme that captures the existence of communication deadlocks. The encoding is based on modelling executions with partial orders, and implemented in a tool called MOPPER. The tool executes an MPI program, collects the trace, builds a formula from the trace using the propositional encoding scheme, and checks its satisfiability. Finally, we present experimental results that quantify the benefit of the approach in comparison to other analysers and demonstrate that it offers a scalable solution for single-path programs.

Additional Key Words and Phrases: single-path verification, MPI programs, deadlock discovery

1. INTRODUCTION

Distributed systems are often developed using the *message passing* paradigm, where the only way to share data between processes is by passing messages over a network. Message passing generally leads to modular, decentralized designs owing to its shared-nothing-by-default model.

The Message Passing Interface (MPI) [Message Passing Interface Forum 2009] is the *lingua franca* of high-performance computing (HPC) and remains one of the most widely used APIs for building distributed message-passing applications.

However, message passing systems are hard to design as they require implementing and debugging complex protocols. These protocols and their interleaved executions are often non-trivial to analyse as the safety and liveness properties of such systems are usually violated only during some intricate, low-probability interleavings. Given the wide adoption of the MPI in large-scale studies in science and engineering, it is important to have means to establish some formal guarantees, like deadlock-freedom, on the behaviour of MPI programs.

In this work, we present an automated method to discover *communication deadlocks* in MPI programs that use blocking and nonblocking (asynchronous) point-to-point communication calls (such as **send** and **receive** calls) and global synchronization primitives (such as barriers). A communication deadlock (referred to simply as "deadlock" in this paper), as described by Natarajan [1984], is "a situation in which each member process of the group is waiting for some member process to communicate with it, but no member is attempting to communicate with it".

Establishing deadlock-freedom in MPI programs is hard. This is primarily due to the presence of nondeterminism that is induced by various MPI primitives and the buffering/arbitration effects in the MPI nodes and the network. For instance, a popular choice in MPI programs to achieve better performance (as noted by Vakkalanka et al. [2008]) is the use of receive calls with the `MPI_ANY_SOURCE` argument; such calls are called "wildcard receives". A wildcard **receive** in a process can be matched with any

[1]Majority of this work was carried out when the author was employed and supported by the Department of Computer Science, University of Oxford.

sender targeting the process, thus the matching between senders and receivers is susceptible to network delivery nondeterminism. MPI calls such as **probe** and **wait** are sources of nondeterminism as well. This prevalence—and indeed, preference—for nondeterminism renders MPI programs susceptible to the schedule-space explosion problem.

Additional complexity in analysing MPI programs is introduced when control-flow decisions are based on random data, or when the data communicated to wildcard receives is used to determine the subsequent control-flow of the program. We call such MPI programs multi-path programs; programs that are not multi-path are called *single-path*. More formally, a program is single-path if for each process, the sequence of instructions executed is the same no matter what data the process receives through MPI calls, and from what processes. We focus on single-path programs in this paper. The rationale for focussing on single-path programs is also found in numerous other domains. For instance, the single-path property is the basis of recent work on verifying GPU kernels [Leung et al. 2012].

Popular MPI debuggers or program correctness checkers such as [Luecke et al. 2002; Hilbrich et al. 2012; Krammer et al. 2003; Haque 2006] only offer limited assistance in discovering deadlocks in programs with wildcard receives. The debuggers/checkers concern themselves exclusively with the send-receive matches that took place in the execution under observation: alternate matches that could potentially happen in the same execution are not explored, nor reasoned about.

On the more formal side, tools such as model checkers can discover bugs related to nondeterministic communication by exploring all relevant matchings/interleavings. However, such tools suffer from several known shortcomings. In some cases, the model has to be constructed manually [Siegel 2007], while some tools have to re-execute the entire program until the problematic matching is discovered [Vakkalanka 2010; Vo et al. 2010]. These limitations prevent such tools from analysing MPI programs that are complex, make heavy use of nondeterminism, or take long to run.

We analyse MPI programs under two different buffering modes: (i) the zero-buffering model, wherein the nodes do not provide buffering and messages are delivered synchronously, and (ii) the infinite-buffering model, under which asynchronously sent messages are buffered without limit. These two models differ in their interpretation of the MPI **wait** event. Under the zero-buffering model, each **wait** call associated with a nonblocking **send** blocks until the message is sent and copied into the address space of the destination process. Under the infinite-buffering model, each **wait** call for a nonblocking **send** returns immediately (see Section 2).

Contribution. This paper presents the following novel results for single-path MPI programs.

(1) First, we demonstrate that even for the restricted class of single-path programs, the problem of deadlock discovery is NP-complete (Section 3).
(2) Second, we present a novel MPI analyser that combines a dynamic verifier with a SAT-based analysis that leverages recent results by Alglave et al. [2013] on propositional encodings of constraints over partial orders.

Our tool operates as follows: the dynamic verifier records an execution trace in the form of a sequence of MPI calls. Then, we extract the per-process *matches-before* partial order on those calls (defined in Section 2), specifying restrictions on the order in which the communication calls may match on an alternative trace. We then construct a sufficiently small over-approximate set of *potential matches* [Sharma et al. 2009] for each **send** and **receive** call in the collected trace. Subsequently, we construct a propositional formula that allows us to determine whether there exists a valid MPI run that respects the matches-before order and yields a deadlock. In our implementation of the propositional encoding, the potentially matching calls are modelled by equality constraints over bit vectors, which facilitates Boolean constraint propagation (BCP) in the SAT solver, resulting in good solving times.

We propose two alternative propositional encodings, both of which can handle the same class of programs. The first encoding is more basic while yielding smaller formula on programs with less communication structure, while the second encoding exploits properties of programs with a star-like communication pattern where a single process receives data from multiple processes through an unbroken series of wildcard **receive** calls. A star-like communication pattern is often employed in situations such as map and reduce where the master process has to perform gather/reduce operations on data after mapping the tasks to workers. Star-like communication allows better utilization of interconnects and leads to increased parallelism (it is straightforward to observe that a receiver employing star-like communication can receive from any sender ready with the result as opposed to waiting to receive a message from a specific sender when no star-like communication is employed). Our proposed encoding for star-like communication outperforms the basic encoding when certain properties are met (explained in detail in Section 4.3). We observe that programs that rely on point-to-point communication primitives frequently adopt star-like communication patterns for performance reasons. An exemplar are the communication patterns observed in arithmetic multigrid solvers that implement the master-worker communication pattern.

Our approach is sound and complete for the class of single-path MPI programs and the buffering models we consider, that is, our tool reports neither false alarms nor misses any deadlock. Our experiments indicate significant speedup compared to the analysis time observed when using ISP, which is a dynamic analyser that enumerates matches explicitly by re-running the program [Vakkalanka et al. 2008]. Note that unlike our tool, ISP is able to handle multi-path programs.

Outline. The paper is organized as follows: We begin by introducing the necessary definitions in Section 2. In Sections 3 and 4 we present the complexity results for the studied problem and give the details of our propositional encodings. Section 5 presents the experimental evaluation of our work. Finally, we discuss the relationship to the related work in Section 6.

A preliminary version of this paper has been presented as a conference publication [Forejt et al. 2014]. In addition to improved presentation and proof, this paper extends the original publication with:

— the encoding for star-like programs, which was not given in Forejt et al. [2014], and
— a more comprehensive evaluation of the tool.

2. PRELIMINARIES

In this section we introduce the necessary definitions and formulate the problem we study in this paper.

2.1. Single-Path Programs

We assume that programs are given as a collection of N processes, denoted by $P_1, \ldots, P_N$. From now on, when N is used, it refers to the number of processes in the program. In order to simplify the problem we consider, we will restrict our analysis to programs in which each process executes its instructions in a fixed order. This is formalized as follows.

Definition 2.1 (*Single-Path Program*). A single-path program is a program in which the order of instructions executed by each of the processes P_i is identical in any run of the program.

We remark that while the per-process instruction order is fixed, the program may still have many different behaviors. In particular, the global schedule of the processes may differ between runs; we do not fix the global ordering of instructions. Deadlocks may abort runs prematurely. Data computed may depend on inputs as long as this does not affect branches.

2.2. MPI Programs

For ease of presentation, we choose an "abstract" definition of MPI programs instead of giving semantics for actual code. Intuitively, for each process, our abstraction disregards all instructions except for MPI calls, and only keeps track of the sequence in which the events are executed. Per process, this sequence is unique due to the definition of single-path programs. We denote the events in process P_i by $a_{i,j}$, where j denotes the index (i.e., the position within the sequence) at which the event occurs. We further use $|P_i|$ for the number of events in process i. As every MPI call gives rise to exactly one event, we use the terms "event" and "MPI call" interchangeably. We define the *per-process order* $\preceq_{po}$ on events as follows: $a_{i,j} \preceq_{po} b_{k,\ell}$ if and only if events $a_{i,j}$ and $b_{k,\ell}$ are from the same process (that is, $i = k$), and the index of a is lower or equal to the index of b (that is, $j \leq \ell$).

The types of MPI calls or events that we permit to occur in an MPI program are as follows.

— A nonblocking (resp. blocking) **send** from P_i to P_j indexed at program location $k \leq |P_i|$ is denoted by $nS_{i,k}(j)$ (resp. $bS_{i,k}(j)$). We write just S when the distinction between a blocking or nonblocking call is not important. The nonblocking calls return immediately.
— Similarly, a nonblocking (resp. blocking) **receive** call, $nR_{i,k}(j)$ (resp. $bR_{i,k}(j)$), indicates that P_i receives a message from P_j. We write just R when the distinction between a blocking or nonblocking call is not important. A wildcard **receive** is denoted by writing $*$ in place of j.
— A blocking **wait** call, which returns on successful completion of the associated nonblocking call, is denoted by $W_{i,k}(h_{i,j})$, where $h_{i,j}$ indicates the index of the associated nonblocking call from P_i. A **wait** call to a nonblocking **receive** will return only if a matching **send** call is present and the message is successfully received in the destination address. By contrast, a **wait** call to a nonblocking **send** will return depending on the underlying buffering model.
— A **wait-all** call $W_{i,k}(h_{i,j_1}, \dots h_{i,j_n})$ can be seen as a syntactic shorthand for n consecutive waits $W_{i,k}(h_{i,j_1}), \dots, W_{i,k}(h_{i,j_n})$.
— We write $B_{i,j}$ for the **barrier** calls in process i. Since **barrier** calls (in a process) synchronise uniquely with a per-process barrier call from each process in the system, all barrier matches are totally ordered. Thus, we use $B_{i,j}(d)$ to denote the **barrier** call issued by the process i that will be part of the d-th system-wide **barrier** call. The process i issuing the **barrier** call blocks until all the other processes also issue the barrier d.

For any of the calls, we replace the program location by "$-$" when it is not relevant.

2.3. Buffering in MPI Programs

According to the standard [Message Passing Interface Forum 2009] a nonblocking **send** is completed as soon as the message is copied out of the sender's address space. Thus,

N:	total number of processes, denoted $P_1, \ldots, P_N$
$\|P\|$:	number of locations in P.
$nS_{i,k}(j)$:	non-blocking send from P_i (at loc. k) to P_j.
$bS_{i,k}(j)$:	blocking send from P_i (at loc. k) to P_j
$nR_{i,k}(j)$:	non-blocking receive by P_i (at loc. k) from P_j, or from any process if $j = *$
$bR_{i,k}(j)$:	blocking receive by P_i (at loc. k) from P_j, or from any process if $j = *$
$W_{i,k}(h_{i,j})$:	wait in process P_i (at loc. k) for a non-blocking call at loc. j to finish
$W_{i,k}(h_{i,j_1}, \ldots h_{i,j_n})$:	as above, but wait for all calls at locations $j_1, \ldots, j_n$
$B_{i,j}(d)$:	d-th barier, P_i blocks at loc. j until all processes issue the associated barrier
$\preceq_{po}$:	per-process order; linear when restricted to a single process
$\preceq_{mo}$:	matches-before order; induced by delivery guarantees of MPI protocol
$Issuable(\langle I, M\rangle)$:	issuable calls, where I and M are issued and matched, resp.
$Matchable(\langle I, M\rangle)$:	matchable calls, where I and M are issued and matched, resp.

Fig. 1. Reference of MPI calls and relevant definitions

under the zero-buffering model the **wait** call will return only after the sent message is successfully received by the receiver since there is no underlying communication subsystem to buffer the message. By contrast, under the infinite-buffering model the sent message is guaranteed to be buffered by the underlying subsystem. We assume, without loss of generality, that message buffering happens immediately after the return of the nonblocking **send**, in which case the associated **wait** call will return immediately.

Let $\mathcal{C}$ be the set of all MPI calls in the program, and $\mathcal{C}_i$ the set of MPI calls in P_i, i.e., the set of MPI calls that P_i may execute (note that some calls might not be executed if there is a deadlock). A *match* is a subset of $\mathcal{C}$ containing those calls that together form a valid communication. A set containing matched **send** and **receive** operations, or a set of matched **barrier** operations, or a singleton set containing a **wait** operation are all matches.

Furthermore, we define a *matches-before* partial order $\preceq_{mo}$, which captures a partial order among communication operations in $\mathcal{C}_i$. We refer the reader to [Vakkalanka et al. 2008] for complete details on the matches-before order (called *completes-before* therein). This order is different for the zero-buffering and infinite-buffering model. For the zero-buffering model, it is defined to be the smallest order satisfying that for any $a, b \in \mathcal{C}$, $a \prec_{mo} b$ if[2] $a \prec_{po} b$ and one of the following conditions is satisfied:

— a is blocking;
— a, b are nonblocking **send** (or **receive**) calls to (from) the same destination (source);
— a is a nonblocking wildcard **receive** call and b is a **receive** call sourcing from P_k (for some k), or a wildcard **receive**;
— a is a nonblocking call and b is the associated **wait** call (or an associated **wait-all** call).

When a is a nonblocking **receive** call sourcing from P_k, b is a nonblocking wildcard **receive** call and the MPI program is at a state where both the calls are issued but not matched yet, then $a \prec_{mo} b$ is *conditionally dependent* on the availability of a matching **send** for a (as noted in [Vakkalanka et al. 2008]). Due to its schedule-dependent nature, we ignore this case in the construction of our encoding. In our experience, we have not come across a benchmark that issues a conditional matches-before edge.

In the case of the infinite-buffering model, the only change is that the rule of the last item in the list above does not apply when a is the nonblocking **send**; this corresponds

[2]We follow standard notation and use $a \prec_{mo} b$ to denote that $a \preceq_{mo} b$ and $a \neq b$. Similarly for $\preceq_{po}$.

to the fact that all nonblocking sends are immediately buffered, and so all the waits for such sends return immediately.

Since the only difference between the finite- and infinite-buffering model is the way the order $\prec_{mo}$ is defined, most of the constructions we present apply for both models. When it is necessary to make a distinction, we will point this out to the reader.

2.4. Semantics of MPI Programs

We now define the behaviour of MPI programs. The current *state* $q = \langle I, M \rangle$ of the system is described by the set of calls I that have been issued, and a set of calls $M \subseteq I$ that were issued and subsequently matched. To formally define a transition system for an MPI program, we need to reason about the calls that can be issued or matched in q. The first is denoted by the set $\mathit{Issuable}(q)$, which is the set of all $x \in \mathcal{C} \setminus I$ satisfying the following two conditions: (i) for all y with $y \prec_{po} x$ we have $y \in I$, and (ii) for all *blocking* z (i.e., any **wait**, **barrier** and blocking **send** and **receive**) with $z \prec_{mo} x$ we have $z \in M$. We say that a set $m \subseteq I \setminus M$ of calls is *ready* in $q = \langle I, M \rangle$ if for every $a \in m$ and every $s \prec_{mo} a$ we have $s \in M$. We then define

$$\begin{aligned}\mathit{Matchable}(q) \;=\; &\{\{a, b\} \text{ ready in } q \mid \exists i, j :\; a = S_{i,-}(j), b = R_{j,-}(i/*)\} \;\cup \\ &\{\{a\} \text{ ready in } q \mid \exists i :\; a = W_{i,-}(h_{i,-})\} \;\cup \\ &\{\{a_1, \cdots, a_N\} \text{ ready in } q \mid \exists d\, \forall i \in [1, N] :\; a_i = B_{i,-}(d)\}\end{aligned}$$

The semantics of an MPI program $\mathcal{P}$ is given by a finite state machine $\mathcal{S}(\mathcal{P}) = \langle \mathcal{Q}, q_0, \mathcal{A}, \delta \rangle$ where

— $\mathcal{Q} \subseteq 2^{\mathcal{C}} \times 2^{\mathcal{C}}$ is the set of states where each state q is a tuple $\langle I, M \rangle$ satisfying $M \subseteq I$, with I being the set of calls that were so far issued by the processes in the program, and M being the set of calls that were already matched.
— $q_0 = \langle \emptyset, \emptyset \rangle$ is the starting state.
— $\mathcal{A} \subseteq 2^{\mathcal{C}}$ is the set of actions.
— $\delta \subseteq \mathcal{Q} \times \mathcal{A} \to \mathcal{Q}$ is the transition function which is the union of two sets of transitions
  — *issue transitions* defined by $\langle I, M \rangle \xrightarrow{\alpha} \langle I \cup \alpha, M \rangle$, if $\alpha \subseteq \mathit{Issuable}(\langle I, M \rangle)$ and $|\alpha|=1$.
  — *match transitions* defined by $\langle I, M \rangle \xrightarrow{\alpha} \langle I, M \cup \alpha \rangle$, if $\alpha \subseteq \mathit{Matchable}(\langle I, M \rangle)$.

The set of *potential matches* $\mathbb{M}$ is defined by $\mathbb{M} = \bigcup_{q \in \Sigma} \mathit{Matchable}(q)$, where $\Sigma \subseteq \mathcal{Q}$ is the set of states that can be reached on some trace starting in q_0. A *trace* is a sequence of states and transitions $q_0 \xrightarrow{\alpha_0} q_1 \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_{n-1}} q_n$ beginning with q_0 such that $q_i \xrightarrow{a_i} q_{i+1}$ for every $0 \leq i < n$.

In a single-path MPI program, all possible execution traces can be generated from a single non-deadlocking execution trace. This is because, following the definition of a single-path MPI program, the set of control-flow paths of the program that contains communication calls is a singleton set. Hence, a single run leads to the execution of *all* the communication calls in the program. Consequently, $\mathbb{M}$ and $\mathcal{Q}$ are sound over-approximations of the possible matches and states of the program.

Example 2.2 (*Running example*). Consider the C MPI program from Figure 2. For three processes, we get a program $\mathcal{P}$ whose MPI calls (forming the set $\mathcal{C}$) are depicted in Figure 3. Then $\preceq_{po}$ and $\preceq_{mo}$ are both equal to the smallest partial order containing (b_1, b_2). Considering a state $q = \langle \{a_1, b_1\}, \emptyset \rangle$ of $\mathcal{S}(\mathcal{P})$, we have $\mathit{Issuable}(q) = \{a_2\}$ and $\mathit{Matchable}(q) = \{\{a_1, b_1\}\}$. The set $\mathbb{M}$ contains elements $\{a_1, b_1\}$, $\{a_2, b_1\}$, and $\{a_1, b_2\}$.

```
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size, sbuff, rbuff;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if(rank == 0) {
    for (int i = 1; i < size-1 ; i++)
      MPI_Recv(&rbuff, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&rbuff, 1, MPI_INT, 2, 1, MPI_COMM_WORLD, &status);
  }
  else
    MPI_Send(&sbuff, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
  MPI_Finalize();
}
```

Fig. 2. The MPI C source code for the running example.

P_1	P_2	P_3
$b_1 : bR_{1,1}(*)$	$a_1 : bS_{2,1}(1)$	$a_2 : bS_{3,1}(1)$
$b_2 : bR_{1,2}(2)$		

Fig. 3. The MPI program for the running example, for three processes.

Some of the traces in $\mathcal{S}(\mathcal{P})$ are deadlocking, for example

$$\langle\emptyset,\emptyset\rangle \xrightarrow{\{a_1\}} \langle\{a_1\},\emptyset\rangle \xrightarrow{\{b_1\}} \langle\{a_1,b_1\},\emptyset\rangle \xrightarrow{\{a_1,b_1\}} \langle\{a_1,b_1\},\{a_1,b_1\}\rangle \xrightarrow{\{a_2\}} \langle\{a_1,b_1,a_2\},\{a_1,b_1\}\rangle \xrightarrow{\{b_2\}} \langle\{a_1,b_1,a_2,b_2\},\{a_1,b_1\}\rangle$$

2.5. The Deadlock Discovery Problem

A state $\langle I, M\rangle$ is *deadlocking* if $M \neq \mathcal{C}$ and it is not possible to make any (issue or match) transition from $\langle I, M\rangle$. A trace is deadlocking if it ends in a deadlocking state. In this paper, we are interested in finding deadlocking traces and the problem we study is formally defined as follows.

Definition 2.3. Given an MPI program $\mathcal{P}$, the *deadlock discovery problem* asks whether there is a deadlocking trace in $\mathcal{S}(\mathcal{P})$.

3. COMPLEXITY OF THE PROBLEM

In this section we prove the following theorem.

THEOREM 3.1. *The deadlock discovery problem is NP-complete, for both the finite- and infinite-buffering model.*

The membership in NP follows easily. All traces are of polynomial size, because after every transition, new elements are added to the set of issued or matched calls, and maximal size of these sets is $|\mathcal{C}|$. Hence, we can guess a sequence of states and actions, and check that they determine a deadlocking trace. This check can be performed in polynomial time, because the partial order $\preceq_{mo}$ can be computed in polynomial time, as well as the sets $\mathit{Issuable}(q)$ and $\mathit{Matchable}(q)$, for any given state q.

Proving the lower bound of Theorem 3.1 is more demanding. We provide a reduction from SAT; the reduction applies to both finite- and infinite-buffering semantics, because it only uses the calls whose semantics is the same under both models. Let Ψ be a

$Ppos_i$	$Pneg_i$	$Pdec_i$	$Pcla_j$	$Pvar$	$Pres$	$Psat$
$bS_{pos_i,1}(dec_i)$	$bS_{neg_i,1}(dec_i)$	$bR_{dec_i,1}(*)$	$bR_{cla_j,1}(*)$	$bS_{var,1}(res)$	$bR_{res,1}(*)$	$bR_{sat,1}(cla_1)$
$\forall c_k \exists x_i$:	$\forall c_k \exists \neg x_i$:	$bS_{dec_i,2}(var)$	$bS_{cla_j,2}(sat)$	$bR_{var,2}(*)$	$bR_{res,2}(sat)$	$\vdots$
$bS_{pos_i,-}(cla_k)$	$bS_{neg_i,-}(cla_k)$	$bR_{dec_i,3}(*)$	$bR_{cla_j,3}(*)$	$\vdots$		$bR_{sat,m}(cla_m)$
			$\vdots$	$bR_{var,n+1}(*)$		$bS_{sat,m+1}(res)$
			$bR_{cla_j,\lvert c_j\rvert+1}(*)$			

Fig. 4. The MPI program $\mathcal{P}(\Psi)$. Here, i ranges from 1 to n and j ranges from 1 to m.

CNF formula over propositional variables $x_1, \ldots, x_n$ with clauses $c_1, \ldots, c_m$, where each clause c_i contains $|c_i|$ literals, we define an MPI program $\mathcal{P}(\Psi)$ as follows. We will create an MPI program with $N = 3 \cdot n + m + 3$ processes. To make the proof easier to read, we introduce auxiliary variables, and use $pos_i = i$, $neg_i = n+i$, $dec_i = 2 \cdot n+i$, $cla_j = 3 \cdot n+j$, $var = 3 \cdot n + j + 1$, $res = 3 \cdot n + j + 2$ and $sat = 3 \cdot n + j + 3$ for $1 \leq i \leq n$ and $1 \leq j \leq m$ to refer to distinct process indices. Communication in process $Ppos_i$ (or $Pneg_i$) will correspond to positive (or negative) values of x_i. The process $Pdec_i$ will ensure that at most one of $Ppos_i$ and $Pneg_i$ can communicate before a certain event, making sure that a value of x_i is simulated correctly.

Further, for each $1 \leq j \leq m$ we create a process $Pcla_j$, and we also create three distinguished processes, $Pvar$, $Pres$ and $Psat$. Hence, the total number of processes is $3 \cdot n + m + 3$.

The communication of the processes is defined in Figure 4. In the figure, the expression $\forall c_k \exists x_i : bS_{pos,-}(cla_k)$ is a shorthand for several consecutive sends, one to each $Pcla_k$ such that $x_i \in c_k$. The order in which the calls are made is not essential for the reduction.

To establish the lower bound for Theorem 3.1, we need to prove the following.

LEMMA 3.2. *A given CNF formula Ψ is satisfiable if and only if the answer to the deadlock discovery problem for $\mathcal{P}(\Psi)$ is yes.*

The crucial observation for the proof of the lemma is that for a deadlock to occur, the call $bS_{sat,m+1}(res)$ must be matched with $bR_{res,1}(*)$: in such a case, the calls $bR_{res,2}(s)$ and $bS_{var,1}(res)$ cannot find any match. In any other circumstance a deadlock cannot occur, in particular note that any $S_{pos_i,-}(cla_k)$, and $S_{neg_i,-}(cla_k)$ can find a matching **receive**, because there are exactly $|c_k|$ sends sent to every $Pcla_k$.

For $bS_{sat,m+1}(res)$ and $bR_{res,1}(*)$ to form a match together, calls $bR_{sat,j}(cla_j)$, $1 \leq j \leq m$, must find a match before $Pvar$ starts to communicate. To achieve this, having a satisfying valuation ν for Ψ, for every $1 \leq i \leq n$ we match $bS_{pos_i,1}(dec_i)$ or $bS_{neg_i,1}(dec_i)$ with $bR_{dec,1}(*)$, depending on whether x_i is true or false under ν. We then match the remaining calls of $Ppos_i$ or $Pneg_i$, and because ν is satisfying, we know that eventually the call $bS_{cla_j,2}(sat)$ can be issued and matched with $bR_{sat,j}(cla_j)$, for all j.

On the other hand, if there is no satisfying valuation for Ψ, then unless for some i both the calls $bS_{pos_i,1}(dec_i)$ and $bS_{neg_i,1}(dec_i)$ find a match, some $bS_{cla_j,2}(sat)$ (and hence also $bR_{sat,j}(cla_j)$) remains unmatched. However, for both $bS_{pos_i,1}(dec_i)$ and $bS_{neg_i,1}(dec_i)$ to match, $bS_{dec_i,2}(var)$ must match some **receive** in $Pvar$, which violates the necessary condition for the deadlock to happen, i.e., that $Pvar$ does not enter into any communication.

Example 3.3. Consider an unsatisfiable formula $(x_1) \wedge (\neg x_1)$. The resulting MPI program is given in Figure 5 (top left and bottom). The MPI program is not deadlocking: intuitively, the reason is that the call $bS_{var,1}(res)$ will be matched before $bS_{sat,3}(res)$. This

$Ppos_1$	$Pneg_1$	$Pdec_1$
$bS_{pos_1,1}(dec_1)$	$bS_{neg_2,1}(dec_1)$	$bR_{dec_1,1}(*)$
$bS_{pos_1,2}(cla_1)$	$bS_{neg_1,2}(cla_2)$	$bS_{dec_1,2}(var)$
		$bR_{dec_1,3}(*)$

$Ppos_1$	$Pneg_1$	$Pdec_1$
$bS_{pos_1,1}(dec_1)$	$bS_{neg_2,1}(dec_1)$	$bR_{dec_1,1}(*)$
$bS_{pos_1,2}(cla_1)$		$bS_{dec_1,2}(var)$
$bS_{pos_1,3}(cla_2)$		$bR_{dec_1,3}(*)$

$Pcla_1$	$Pcla_2$	$Pvar$	$Pres$	$Psat$
$bR_{cla_1,1}(*)$	$bR_{cla_2,1}(*)$	$bS_{var,1}(res)$	$bR_{res,1}(*)$	$bR_{sat,1}(cla_1)$
$bS_{cla_1,2}(sat)$	$bS_{cla_2,2}(sat)$	$bR_{var,2}(*)$	$bR_{res,2}(sat)$	$bR_{sat,2}(cla_2)$
		$bR_{var,3}(*)$		$bS_{sat,3}(res)$

Fig. 5. Examples of the reduction for the formulas $(x_1) \wedge (\neg x_1)$ (top left and bottom) and $(x_1) \wedge (x_1)$ (top right and bottom).

is because for $bS_{sat,3}(res)$ to match, both $bR_{cla_1,1}(*)$ and $bR_{cla_2,1}(*)$ must be matched, which requires that $bS_{var,1}(res)$ is matched.

On the other hand, a satisfiable formula $(x_1) \wedge (x_1)$ yields the program from Figure 5 (top right and bottom), which can deadlock by first all calls of $Ppos_1$ finding a match, and then all calls of $Psat$ finding a match. This results in $bR_{res,2}(sat)$ not being able to find any match.

4. DEADLOCK DISCOVERY USING SAT

4.1. Propositional Encoding

In this section we introduce a propositional encoding for solving the deadlock discovery problem. Intuitively, a satisfying valuation for the variables in the encoding provides a set of calls matched on a trace, a set of unmatched calls that can form a match, and a set of matches together with a partial order on them, which contains enough dependencies to ensure that the per-process partial order is satisfied.

We will restrict the presentation to the problem without barriers, since barriers can be removed by preprocessing, where for barrier calls $B_{i,-}(d)$ and $B_{j,-}(d)$ and for any two calls a and b such that $a \prec_{mo} B_{i,-}(d)$ and $B_{j,-}(d) \prec_{mo} b$ we assume $a \prec_{mo} b$. The barrier calls can then be removed without introducing spurious models.

Our encoding contains variables m_a and r_a for every call a. Their intuitive meaning is that a is matched or issued (“ready”) to be matched whenever m_a or r_a is true, respectively. Supposing we correctly identify the set of matched and issued calls on a trace, we can determine whether a deadlock has occurred. For this to happen, there must be some unmatched call, and no potential match can take place (i.e., for any potential match, some call was either used in another match, or was not issued yet). Thus, we must ensure that we determine the matched and issued calls correctly. We impose a preorder on the calls, where a occurs before b in the preorder if a finds a match before b. To capture the preorder, we use the variables t_{ab} to denote that a matches before b, and s_{ab} which stipulate that a call a matches a **receive** b and hence they must happen at the same time; note that this applies in the infinite-buffering case as well.

Finally, we must ensure that t_{ab} and s_{ab} correctly impose a preorder. We use a bit vector clk_a of size $\lceil \log_2 |\mathcal{C}| \rceil$ for every call a, denoting the “time” at which the call a happens, and stipulate that $clk_a < clk_b$ (resp. $clk_a = clk_b$) if t_{ab} (resp. s_{ab}) is true. Intuitively, clk_a represents a time that the call a finds its matching call. Nevertheless, this number might not exactly correspond to a position of a match in a trace. This is because our encoding admits satisfying assignments which are not a total order (i.e., intuitively, two matches can happen at the same time), and in which there are “gaps” between two matching events.

As part of the input, our encoding requires a set $\mathbb{M}^+ \supseteq \mathbb{M}$ containing sets of calls that are type-compatible (i.e., all α that can be contained in some $Matchable(q)$ if we

Partial order
$$\bigwedge_{b \in \mathcal{C}} \bigwedge_{a \in Imm(b)} t_{ab} \tag{1}$$

Unique match for **send**
$$\bigwedge_{(a,b) \in \mathbb{M}^+} \bigwedge_{c \in \mathbb{M}^+(a), c \neq b} \big(s_{ab} \rightarrow \neg s_{ac}\big) \tag{2}$$

Unique match for **receive**
$$\bigwedge_{(a,b) \in \mathbb{M}^+} \bigwedge_{c \in \mathbb{M}^+(b), c \neq a} \big(s_{ab} \rightarrow \neg s_{cb}\big) \tag{3}$$

Match correct
$$\bigwedge_{a \in \mathsf{R}} \big(m_a \rightarrow \bigvee_{b \in \mathbb{M}^+(a)} s_{ba}\big) \wedge \bigwedge_{a \in \mathsf{S}} \big(m_a \rightarrow \bigvee_{b \in \mathbb{M}^+(a)} s_{ab}\big) \tag{4}$$

Matched only
$$\bigwedge_{\alpha \in \mathbb{M}^+} \big(s_\alpha \rightarrow \bigwedge_{a \in \alpha} m_a\big) \tag{5}$$

No match possible
$$\bigwedge_{\alpha \in \mathbb{M}^+} \big(\bigvee_{a \in \alpha} (m_a \vee \neg r_a)\big) \tag{6}$$

All ancestors matched
$$\bigwedge_{b \in \mathcal{C}} \big(r_b \leftrightarrow \bigwedge_{a \in Imm(b)} m_a\big) \tag{7}$$

Not all matched
$$\bigvee_{a \in \mathcal{C}} \neg m_a \tag{8}$$

Match only issued
$$\bigwedge_{a \in \mathcal{C}} \big(m_a \rightarrow r_a\big) \tag{9}$$

Clock equality
$$\bigwedge_{(a,b) \in \mathbb{M}^+} \big(s_{ab} \rightarrow (clk_a = clk_b\big) \tag{10}$$

Clock difference
$$\bigwedge_{a,b \in \mathcal{C}} \big(t_{ab} \rightarrow (clk_a < clk_b)\big) \tag{11}$$

Fig. 6. The SAT encoding for the deadlock discovery. Here, empty conjunctions are true and empty disjunctions are false.

disregard the requirement for α to be ready). The reason for not starting directly with $\mathbb{M}$ is that the problem of deciding whether a given set α is a potential match, i.e., whether $\alpha \in \mathbb{M}$, is NP-complete. This result can be obtained as a simple corollary of our construction for Lemma 3.2. Hence, in any practical implementation we must start with $\mathbb{M}^+$, since computing the set $\mathbb{M}$ is as hard as the deadlock discovery problem itself. We will give a reasonable candidate for $\mathbb{M}^+$ in the next section.

The formal definition of the encoding is presented in Figure 6. In the figure, S and R are the sets containing all **send** and **receive** calls, respectively, $Imm(a) = \{x | x \prec_{mo} a, \forall z : x \preceq_{mo} z \preceq_{mo} a \Rightarrow z \in \{x, a\}\}$ stands for the set of immediate predecessors of a, and $\mathbb{M}^+(a) = \bigcup\{b \mid \exists \alpha \in \mathbb{M}^+ : a, b \in \alpha\} \setminus \{a\}$ is the set of all calls with which a can form a match. Further, $clk_a = clk_b$ (resp. $clk_a < clk_b$) are shorthands for the formulas that are true if and only if the bit vector for a encodes the value equal to (resp. lower than) the value of the bit vector for b. The formula constructed contains $\mathcal{O}(|\mathcal{C}|^2)$ variables, and its size is in $\mathcal{O}(|\mathcal{C}|^3)$.

Example 4.1. Consider the example program $\mathcal{P}$ from Example 2.2. The formula resulting from the program is given in Figure 7 (with part of constraint (11) omitted). For all calls c, the value clk_c is represented in the little-endian format as $x_{c,1}x_{c,2}$. The reader can easily verify that the assignment that assigns true to exactly the following

(1): $t_{b_1 b_2}$
(2): $(s_{a_1 b_1} \rightarrow \neg s_{a_1 b_2}) \wedge (s_{a_1 b_2} \rightarrow \neg s_{a_1 b_1})$
(3): $(s_{a_1 b_1} \rightarrow \neg s_{a_2 b_1}) \wedge (s_{a_2 b_1} \rightarrow \neg s_{a_2 b_1})$
(4): $\big(m_{a_1} \rightarrow (s_{a_1 b_1} \vee s_{a_1 b_2})\big) \wedge \big(m_{a_2} \rightarrow s_{a_2 b_1}\big) \wedge \big(m_{b_1} \rightarrow (s_{a_1 b_1} \vee s_{a_2 b_1}\big) \wedge \big(m_{b_2} \rightarrow s_{a_1 b_2}\big)$
(5): $\big(s_{a_1 b_1} \rightarrow (m_{a_1} \wedge m_{b_1})\big) \wedge \big(s_{a_1 b_2} \rightarrow (m_{a_1} \wedge m_{b_2})\big) \wedge \big(s_{a_2 b_1} \rightarrow (m_{a_2} \wedge m_{b_1})\big)$
(6): $(m_{a_1} \vee \neg r_{a_1} \vee m_{b_1} \vee \neg r_{b_1}) \wedge (m_{a_1} \vee \neg r_{a_1} \vee m_{b_2} \vee \neg r_{b_2}) \wedge (m_{a_2} \vee \neg r_{a_2} \vee m_{b_1} \vee \neg r_{b_1})$
(7): $r_{b_2} \leftrightarrow m_{b_1}$
(8): $\neg m_{a_1} \vee \neg m_{a_2} \vee \neg m_{b_1} \vee \neg m_{b_2}$
(9): $(m_{a_1} \rightarrow r_{a_1}) \wedge (m_{a_2} \rightarrow r_{a_2}) \wedge (m_{b_1} \rightarrow r_{b_1}) \wedge (m_{b_2} \rightarrow r_{b_2})$
(10): $\Big(s_{a_1 b_1} \rightarrow \big((x_{a_1,1} \leftrightarrow x_{b_1,1}) \wedge (x_{a_1,2} \leftrightarrow x_{b_1,2})\big)\Big) \wedge \Big(s_{a_1 b_2} \rightarrow \big((x_{a_1,1} \leftrightarrow x_{b_2,1})$
$\wedge (x_{a_1,2} \leftrightarrow x_{b_2,2})\big)\Big) \wedge \Big(s_{a_2 b_1} \rightarrow \big((x_{a_2,1} \leftrightarrow x_{b_1,1}) \wedge (x_{a_2,2} \leftrightarrow x_{b_1,2})\big)\Big)$
(11): $t_{b_1 b_2} \rightarrow \Big((\neg x_{b_1,1} \wedge x_{b_2,1}) \vee \big((x_{b_1,1} \leftrightarrow x_{b_2,1}) \wedge (\neg x_{b_1,2} \wedge x_{b_2,2})\big)\Big)$ [. . .]

Fig. 7. Propositional encoding for the running example.

propositions is satisfying: $t_{b_1 b_2}$, $s_{a_1 b_1}$, m_{a_1}, m_{b_1}, r_{a_1}, r_{a_2}, r_{b_1}, r_{b_2}, and $x_{b_2,2}$. This means that there is a deadlocking trace.

4.2. Correctness of the Encoding

The correctness of the encoding is formally established by Lemmas 4.2 and 4.4.

LEMMA 4.2. *For every deadlocking trace there is a satisfying assignment to the variables in the encoding.*

PROOF. Given a deadlocking trace, we construct the satisfying assignment as follows. We set m_a to true if and only if a is matched on the trace, and r_a true if and only if it is matched or if for every $b \prec_{mo} a$, m_b is true. This makes sure the constraints (6)–(9) are satisfied.

We assign s_{ab} to true if and only if $\{a, b\}$ occurs as a match on the trace. This ensures satisfaction of constraints (2)–(5). Further, let $\alpha_1 \alpha_2 \ldots$ be the sequence of actions under which match transitions are taken on the trace. We stipulate t_{ab} if $a \in \alpha_i$ and $b \in \alpha_j$ for $i < j$. We also set $clk_a = i$ for every $a \in \alpha_i$ and every i. This ensures satisfaction of the remaining constraints. □

The following lemma follows easily from constraints (2) and (3).

LEMMA 4.3. *In every satisfying assignment to the variables in the encoding we have that for every a, if s_{ab} and $s_{ab'}$ are true, then $b = b'$, and also if s_{ba} and $s_{b'a}$ are true, then $b = b'$.*

LEMMA 4.4. *For every satisfying assignment to the variables in the encoding there is a deadlocking trace.*

PROOF. Given a satisfying assignment, we construct the trace as follows. Let A be the set of all sends and waits such that $a \in A$ if and only if m_a is true, and let $a_1 \ldots a_K$ be an ordered sequence of elements in A such that for any a_i and a_j, if $clk_{a_i} < clk_{a_j}$, then $i < j$. We further define a sequence $\theta = \alpha_1 \ldots \alpha_K$, where every α_i contains a_i, and if a_i is a **send**, then α_i also contains the unique **receive** b_i such that $s_{a_i b_i}$ is true. Such b_i always exists, and is unique by Lemma 4.3. By (10) the sequence θ satisfies that whenever $a \in \alpha_i$ and $b \in \alpha_j$ and $clk_a < clk_b$, then $i < j$. Moreover, for any c we have that the proposition m_c is true if and only if c occurs in some α_i; this follows by the construction of A and by (4) and (5).

We define a trace from the sequence θ by stipulating that it visits the states

$$q_i = \langle I_i, M_i \rangle = \langle\ \{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \le \ell \le i} \alpha_\ell\}\ ,\ \bigcup_{1 \le \ell \le i} \alpha_\ell\ \rangle$$

for $0 \le i \le K$. The part of the trace from q_i to q_{i+1} is defined to be

$$q_i \xrightarrow{\{b_{i,1}\}} \langle I_i \cup \{b_{i,1}\}, M_i \rangle \xrightarrow{\{b_{i,2}\}} \dots \xrightarrow{\{b_{i,n_i}\}} \langle I_i \cup \{b_{i,1}, \dots b_{i,n_i}\}, M_i \rangle \xrightarrow{\alpha_{i+1}} q_{i+1}$$

where all except for the last transition are issue transitions, where $\{b_{i,1}, \dots, b_{i,n_i}\} = \{y \mid \exists x \succeq_{po} y : x \in \alpha_{i+1}\} \setminus \{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \le \ell \le i} \alpha_\ell\}$, and where if $b_{i,j} \prec_{po} b_{i,\ell}$, then $j < \ell$.

We now argue that the sequence above is indeed a valid trace in $\mathcal{S}(\mathcal{P})$. Firstly, $q_0 = \langle \emptyset, \emptyset \rangle$. Let i be the largest number such that the sequence from q_0 up to q_i is a valid trace. Let j be the largest number such that the extension of this trace from q_i up to $\langle I, M \rangle = \langle I_i \cup \{b_{i,1}, \dots b_{i,j}\}, M_i \rangle$ is a valid trace. We analyse the possible values of j, showing that each leads to a contradiction.

— Suppose $0 \le j < n_i$. First, note that $b_{i,j+1} \notin I \cup M$, because $b_{i,j+1}$ does not occur in $\{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \le \ell \le i} \alpha_\ell\}$. We need to show that $b_{i,j+1} \in \mathit{Issuable}(\langle I, M \rangle)$ (see page 6 for the definition). Consider any $a \in \mathcal{C}$.
  If $a \prec_{po} b_{i,j+1}$, then by the definition of the sequence $b_{i,1}, \dots b_{i,n_i}$ the element a has been issued already.
  If $a \prec_{mo} b_{i,j+1}$ and a is blocking, then by the definition of $\prec_{mo}$ we have $a \prec_{mo} c$ for $c \in \alpha_{i+1}$ such that $b_{i,j+1} \prec_{po} c$. By applying (1) and (11), possibly multiple times, we establish that $\mathit{clk}_a < \mathit{clk}_c$. Further, for any x and y we have that if r_x or m_x is true and $y \prec_{mo} x$, then m_y is true; this follows from constraints (7) and (9). Hence, m_a is true, and so a must be contained in some α_k for $k < i + 1$.
  Consequently, $b_{i,j+1} \in \mathit{Issuable}(\langle I, M \rangle)$.
— Suppose $j = n_i$. We have argued above that for every element $b \in \alpha_{i+1}$ and every $a \prec_{mo} b$ we have $a \in M$. Also, $b \in I \setminus M$, and so α_{i+1} is ready in $\langle I, M \rangle$. Finally, we defined α_{i+1} to be either a singleton set containing a wait, or a set containing compatible **send** and **receive**, hence, $\alpha_{i+1} \in \mathit{Matchable}(\langle I, M \rangle)$.

Finally, we argue that the trace is deadlocking. By (8) and the construction of the sequence θ we have that $M_K \subsetneq \mathcal{C}$. We show that from $q_K = \langle I_K, M_K \rangle$ it is not possible to make a match transition, even after possibly making a number of issue transitions. This proves that there is a deadlocking trace. Suppose that it is possible to make a match transition, and let us fix a suffix $q_K \xrightarrow{\{b_1\}} \hat{q}_1 \xrightarrow{\{b_2\}} \hat{q}_2 \dots \xrightarrow{\{b_n\}} \hat{q}_n \xrightarrow{\alpha} \bar{q}$ where all transitions except for the last one are issue transitions. Note that because $\hat{q}_n = \langle I_K \cup \{b_1, \dots, b_n\}, M_K \rangle$, for the transition under α to exist it must be the case that for any $b \in \alpha$ and any $a \prec_{mo} b$ we have $a \in M_K$. But then by (7) all $b \in \alpha$ satisfy that r_b is true, and by (6) we get that there is $b \in \alpha$ for which m_b is true, and so $b \in M_K$, which contradicts that the match transition under α can be taken in $\hat{q}_n$. $\square$

4.3. Alternative Propositional Encoding for Programs with Many Wildcard Receives

Our case-studies show that a common pattern in MPI programs is that a designated process receives instructions from multiple other processes using wildcard **receive** calls. To improve the performance for this scenario, we give an alternative propositional encoding. The basic idea of the encoding is that for consecutive wildcard receives in one process, it is not necessary to know which **send** was matched to each respective **receive**, but instead it is sufficient to know the *set of sends* that together match the consecutive receives.

We define a *multi-receive* x to be a maximal set of $bR(*)$ satisfying the following:

— all elements of x are from the same process.
— for every $a, b \in x$ and $c \notin x$ we have that if $a \prec_{mo} c$ (resp. $c \prec_{mo} a$), then $b \prec_{mo} c$ (resp. $c \prec_{mo} b$). Intuitively, multi-receive x represents a maximal contiguous sequence of wildcard receives in a process.

We say that a multi-receive is *matched* if all elements of x are matched.

Let $\mathcal{R}$ be a set of all multi-receives. We use $\bar{\mathcal{C}} = \mathcal{C} \setminus \bigcup \mathcal{R}$ for all calls that are not part of any multi-receive.

We also need to modify the function Imm; for this purpose, we define a function $Imm_{\mathcal{R}}$ where for all $b \in \mathcal{R} \cup \bar{\mathcal{C}}$ we define $Imm_{\mathcal{R}}(b)$ to contain all a such that one of the following holds:

— $a, b \in \bar{\mathcal{C}}$ such that $a \in Imm(b)$
— $a \in \mathcal{R}$, $b \in \bar{\mathcal{C}}$ and for some $c \in a$ we have $c \in Imm(b)$
— $a \in \bar{\mathcal{C}}$, $b \in \mathcal{R}$ and for some $d \in b$ we have $a \in Imm(d)$
— $a, b \in \mathcal{R}$ and for some $c \in a$ and $d \in b$ we have $c \in Imm(d)$

We similarly extend $\mathbb{M}^+$ to $\mathbb{M}^+_{\mathcal{R}}$ as follows: we put to $\mathbb{M}^+_{\mathcal{R}}$ all sets from $\mathbb{M}^+$ that only contain elements of $\bar{\mathcal{C}}$, and also, for every $\{a, b\} \in \mathbb{M}^+$ with $b \in c$ for some $c \in \mathcal{R}$ we put $\{a, c\}$ to $\mathbb{M}^+_{\mathcal{R}}$.

For the alternative encoding, we use atomic propositions m_a, r_a, t_{ab} for all $a, b \in \mathcal{R} \cup \bar{\mathcal{C}}$. We also use one bit-vector clk_a for every $a \in \bar{\mathcal{C}}$, and two bit-vectors, clk^-_a and clk^+_a, for each $a \in \mathcal{R}$ to indicate the time when the first and the last event of the multi-receive a matched, respectively. To simplify the notation, for $a \in \bar{\mathcal{C}}$ we sometimes (e.g., in constraint (24)) refer to clk_a by clk^+_a or clk^-_a. We also introduce an atomic proposition s_{ab} for all $a \in \mathsf{S}$ and $b \in \mathcal{R} \cup \bar{\mathcal{C}}$ such that $\{a, b\} \in \mathbb{M}^+_{\mathcal{R}}$.

Since the encoding is based on matching several sends to a single multi-receive, we need to introduce constraints that enforce the correct number of sends to be matched. To this extent, we define $card(a)$ to be 1 for every $a \in \bar{\mathcal{C}}$, and $|a|$ for every $a \in \mathcal{R}$. With $atmost(k, Z)$ (resp. $exactly(k, Z)$) where Z is a set of propositional variables, we denote propositional constraints that are true if and only if at most (resp. exactly) k many propositional variables from the set Z are true. These constraints are called *cardinality constraints*. There is a variety of propositional encodings for cardinality constraints, see [Bailleux and Boufkhad 2003; **?**] for an overview.

The full propositional encoding for programs with wildcard receives is presented in Figure 8. As in the original encoding, the formula constructed contains $\mathcal{O}(|\mathcal{C}|^2)$ variables, and its size is in $\mathcal{O}(|\mathcal{C}|^3)$.

Example 4.5. We illustrate the approach on the running example. In order to show the details of the encoding, we consider the case when $N = 4$. The program is given in the following table:

P_1	P_2	P_3	P_4
$b_1 : bR_{1,1}(*)$	$a_1 : bS_{2,1}(1)$	$a_2 : bS_{3,1}(1)$	$a_3 : bS_{4,1}(1)$
$b_2 : bR_{1,2}(*)$			
$b_3 : bR_{1,3}(2)$			

There is a single multi-receive, $x = \{b_1, b_2\}$. The atomic propositions we use are

— t_{xb_3}, s_{a_1x}, s_{a_2x}, s_{a_3x}, $s_{a_1b_3}$, and
— m_y and r_y for $y \in \{a_1, a_2, a_3, x, b_3\}$
— variables for bit-vectors (of size 3) clk_{a_i} for $1 \leq i \leq 3$, clk_{b_3}, clk^-_x and clk^+_x.
— auxiliary variables for encoding the *atmost* and *exactly* constraints.

Partial order *
$$\bigwedge_{b \in \mathcal{R} \cup \bar{\mathcal{C}}} \bigwedge_{a \in Imm_{\mathcal{R}}(b)} t_{ab} \tag{12}$$

Maximal match count
$$\bigwedge_{b \in \mathcal{R} \cup \bar{\mathcal{C}}} atmost\big(card(b), \{s_{ab} \mid a \in \mathbb{M}^{+}_{\mathcal{R}}(b)\}\big) \tag{13}$$

$$\bigwedge_{a \in \mathcal{R} \cup \bar{\mathcal{C}}} atmost\big(card(a), \{s_{ab} \mid b \in \mathbb{M}^{+}_{\mathcal{R}}(a)\}\big) \tag{14}$$

Match correct
$$\bigwedge_{b \in \mathcal{R} \cup \bar{\mathcal{C}}} \Big(m_b \leftrightarrow exactly\big(card(b), \{s_{ab} \mid a \in \mathbb{M}^{+}_{\mathcal{R}}(b)\}\big)\Big) \tag{15}$$

$$\bigwedge_{a \in \mathcal{R} \cup \bar{\mathcal{C}}} \Big(m_a \leftrightarrow exactly\big(card(a), \{s_{ab} \mid b \in \mathbb{M}^{+}_{\mathcal{R}}(a)\}\big)\Big) \tag{16}$$

No match possible *
$$\bigwedge_{\alpha \in \mathbb{M}^{+}_{\mathcal{R}}} \big(\bigvee_{a \in \alpha} (m_a \vee \neg r_a)\big) \tag{17}$$

All ancestors matched *
$$\bigwedge_{b \in \mathcal{R} \cup \bar{\mathcal{C}}} \big(r_b \leftrightarrow \bigwedge_{a \in Imm_{\mathcal{R}}(b)} m_a\big) \tag{18}$$

Not all matched *
$$\bigvee_{a \in \mathcal{R} \cup \bar{\mathcal{C}}} \neg m_a \tag{19}$$

Match only issued *
$$\bigwedge_{a \in \mathcal{R} \cup \bar{\mathcal{C}}} \big(m_a \rightarrow r_a\big) \tag{20}$$

$$\bigwedge_{(a,b) \in \mathbb{M}^{+}_{\mathcal{R}}} \big(s_{ab} \rightarrow (r_a \wedge r_b)\big) \tag{21}$$

Clock equality for $\bar{\mathcal{C}}$ *
$$\bigwedge_{(a,b) \in \mathbb{M}^{+}_{\mathcal{R}} \cap (\mathsf{S} \times (\mathsf{R} \setminus \mathcal{R}))} \big(s_{ab} \rightarrow (clk_a = clk_b)\big) \tag{22}$$

Clock equality for $\mathcal{R}$
$$\bigwedge_{(a,b) \in \mathbb{M}^{+}_{\mathcal{R}} \cap (\mathsf{S} \times \mathcal{R})} \big(s_{ab} \rightarrow (clk^{-}_{b} \leq clk_a \leq clk^{+}_{b})\big) \tag{23}$$

Clock difference *
$$\bigwedge_{a,b \in \mathcal{R} \cup \bar{\mathcal{C}}} \big(t_{ab} \rightarrow (clk^{+}_{a} < clk^{-}_{b})\big) \tag{24}$$

Fig. 8. The alternative SAT encoding for the deadlock discovery. The rules taken marked with '*' also occur in the encoding in Figure 6, or are a straightforward modification.

We highlight some of the parts that differ significantly from the previous encoding (for space reasons we keep the abbreviations for *atmost*, *exactly* and bit-vector comparison):
(13): $atmost(2, \{s_{a_1x}, s_{a_2x}, s_{a_3x}\})$
(14): $atmost(1, \{s_{a_1x}, s_{a_1b_1}\}) \wedge atmost(1, \{s_{a_2x}\}) \wedge atmost(1, \{s_{a_3x}\})$
(15): $m_x \leftrightarrow exactly(2, \{s_{a_1x}, s_{a_2x}, s_{a_3x}\})$
(16): $\big(m_{a_1} \leftrightarrow exactly(1, \{s_{a_1x}, s_{a_1b_1}\})\big) \wedge \big(m_{a_2} \leftrightarrow exactly(1, \{s_{a_2x}\})\big) \wedge \big(m_{a_3} \leftrightarrow exactly(1, \{s_{a_3x}\})\big)$
(23): $\big(s_{a_1x} \rightarrow (clk^{-}_{x} \leq clk_{a_1} \leq clk^{+}_{x})\big) \wedge \big(s_{a_2x} \rightarrow (clk^{-}_{x} \leq clk_{a_2} \leq clk^{+}_{x})\big) \wedge \big(s_{a_3x} \rightarrow (clk^{-}_{x} \leq clk_{a_3} \leq clk^{+}_{x})\big)$

4.4. Correctness of the Alternative Encoding

The correctness of the encoding is formally established by Lemmas 4.6 and 4.8. The proofs of the lemmas are essentially straightforward extensions of proofs of Lemmas 4.2

and 4.4, but we present them for the sake of completeness and because they provide insight into intuitions behind the encoding.

LEMMA 4.6. *For every deadlocking trace there is a satisfying assignment to the variables in the encoding from Figure 8.*

PROOF. Given a deadlocking trace, we construct the satisfying assignment as follows. We set m_a to true if and only if a is matched on the trace, and r_a true if and only if it is matched or if for every $b \in \mathit{Imm}_{\mathcal{R}}(a)$, the propositional variable m_b is true. This ensures that the constraints (17)–(20) are satisfied.

We assign s_{ab} to true if and only if:

— $a, b \in \bar{\mathcal{C}}$ and a matches b on the trace, or
— $a \in \bar{\mathcal{C}}$ and $b \in \mathcal{R}$ and a matches an element of b on the trace.

This ensures satisfaction of constraints (13)–(16) and (21).

Further, let $\alpha_1\alpha_2\ldots$ be the sequence of actions under which match transitions are taken on the trace. For every i and $a \in \alpha_i \cap \bar{\mathcal{C}}$ we set $\mathit{clk}_a = i$, and for $a \in \alpha_i \cap \bigcup \mathcal{R}$ and $x \ni a$ we set $\mathit{clk}^-_x = \min\{j \mid \exists b \in x : b \in \alpha_j\}$ and $\mathit{clk}^+_x = \max\{j \mid \exists b \in x : b \in \alpha_j\}$. This ensures satisfaction of the constraints (22) and (23).

We set t_{ab} to true if and only if:

— $a, b \in \bar{\mathcal{C}}$ and $a \in \alpha_i$ and $b \in \alpha_j$ for $i < j$;
— $a \in \bar{\mathcal{C}}$, $b \in x$ for some $x \in \mathcal{R}$, and $a \in \alpha_i$ for $i < \min\{j \mid \exists c \in x : c \in \alpha_j\}$;
— $a \in x$ for some $x \in \mathcal{R}$ and $b \in \bar{\mathcal{C}}$ where $b \in \alpha_i$ for $i > \max\{j \mid \exists c \in x : c \in \alpha_j\}$; or
— $a \in x$ for some $x \in \mathcal{R}$ and $b \in y$ for some $y \in \mathcal{R}$ where $\max\{j \mid \exists c \in x : c \in \alpha_j\} < \min\{j \mid \exists c \in y : c \in \alpha_j\}$.

This ensures satisfaction of constraint (24), and also of constraint (12) due to the definition of $\mathit{Imm}_{\mathcal{R}}$. □

To simplify the notation, for an element a contained in some $x \in \mathcal{R}$ we say that r_a (resp. m_a) is true instead of saying that r_x (resp. m_x) is true.

LEMMA 4.7. *In any satisfying assignment to the variables in the encoding from Figure 8, and for any a and b such that $a \prec_{po} b$ the following holds. If r_b is true, then r_a is also true.*

PROOF. The proof follows from constraints (17) and (20). □

LEMMA 4.8. *For every satisfying assignment to the variables in the encoding from Figure 8 there is a deadlocking trace.*

PROOF. Given a satisfying assignment, we construct the trace as follows. Let A be the set of all sends and waits such that $a \in A$ if and only if m_a is true, and let $a_1 \ldots a_K$ be an ordered sequence of elements in A such that for any a_i and a_j, if $\mathit{clk}_{a_i} < \mathit{clk}_{a_j}$, then $i < j$. We further define a sequence $\theta = \alpha_1 \ldots \alpha_K$ inductively by letting α_i contain a_i and in addition if a_i is a **send**, then for the unique b_i such that $s_{a_i b_i}$ is true:

— if b_i is a **receive**, then α_i also contains b_i.
— if b_i is a multi-receive, then α_i also contains an element of b_i that hasn't been matched before. Formally, let $\leq_{b_i}$ be an arbitrary but fixed total order of elements of b_i which satisfies that whenever $x \prec_{mo} y$, then $x <_{b_i} y$. Then α_i contains the lowest element (with respect to $\leq_{b_i}$) of $b_i \setminus \bigcup_{j<i} \alpha_j$. Existence of such an element follows from (13) and (14).

Note that such b_i always exists and is unique by (15) and (16). The following claims hold true for the sequence θ and all i and j:

(a) For all $a \in \alpha_i \cap \bar{\mathcal{C}}$ and $b \in \alpha_j \cap \bar{\mathcal{C}}$, if $clk_a < clk_b$, then $i < j$. Indeed, we can take the a_i and a_j used to define θ, and observe that $clk_a = clk_{a_i}$ (since either $a = a_i$, or $a \neq a_i$ in which case $s_{a_i a}$ is true and constraint (22) applies) and similarly $clk_b = clk_{b_i}$. Then the rest follows since if $clk_{a_i} < clk_{a_j}$, then $i < j$.

(b) For all $a \in \alpha_i \cap \bar{\mathcal{C}}$ and $b \in \alpha_j \cap x$ for some $x \in \mathcal{R}$, if $clk_a < clk^-_x$, then $i < j$. Indeed, as above we have $clk_a = clk_{a_i}$; we also take a_j and note that by (23) we have $clk^-_x \leq clk_{a_j}$. Hence, $clk_{a_i} < clk_{a_j}$ and so $i < j$.

(c) For all $a \in \alpha_i \cap x$ for some $x \in \mathcal{R}$ and $b \in \alpha_j \cap \bar{\mathcal{C}}$, if $clk^+_x < clk_b$, then $i < j$. Indeed, as in (a) we have $clk_b = clk_{a_j}$; we also take a_i and note that by (23) we have $clk_{a_j} \leq clk^+_x$. Hence, $clk_{a_i} < clk_{a_j}$ and so $i < j$.

(d) For all $a \in \alpha_i \cap x$ for some $x \in \mathcal{R}$ and $b \in \alpha_j \cap y$ for some $y \in \mathcal{R}$, if $clk^+_x < clk^-_y$, then $i < j$. Indeed, as above, we consider that by (23) we have $clk_{a_i} \leq clk^+_x$ and $clk^-_y \leq clk_{a_j}$, and so $i < j$.

Moreover for all $x \in \bar{\mathcal{C}}$ we have that if m_x is true, then $x \in \bigcup_{i=1}^{K} \alpha_i$, and for all $x \in \mathcal{R}$ we have that if m_x is true, then $x \subseteq \bigcup_{i=1}^{K} \alpha_i$; this follows by the construction of A and by (15) and (16).

We define a trace from the sequence θ exactly as in Lemma 4.4. We stipulate that the trace visits the states

$$q_i = \langle I_i, M_i \rangle = \langle \ \{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \leq \ell \leq i} \alpha_\ell\} \ , \ \bigcup_{1 \leq \ell \leq i} \alpha_\ell \ \rangle$$

for $0 \leq i \leq K$. The part of the trace from q_i to q_{i+1} is defined to be

$$q_i \xrightarrow{\{b_{i,1}\}} \langle I_i \cup \{b_{i,1}\}, M_i \rangle \xrightarrow{\{b_{i,2}\}} \ldots \xrightarrow{\{b_{i,n_i}\}} \langle I_i \cup \{b_{i,1}, \ldots b_{i,n_i}\}, M_i \rangle \xrightarrow{\alpha_{i+1}} q_{i+1}$$

where all except for the last transition are issue transitions, where $\{b_{i,1}, \ldots, b_{i,n_i}\} = \{y \mid \exists x \succeq_{po} y : x \in \alpha_{i+1}\} \setminus \{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \leq \ell \leq i} \alpha_\ell\}$, and where if $b_{i,j} \prec_{po} b_{i,\ell}$, then $j < \ell$.

We now argue that the sequence above is indeed a valid trace in $\mathcal{S}(\mathcal{P})$. Firstly, $q_0 = \langle \emptyset, \emptyset \rangle$. Let i be the largest number such that the sequence from q_0 up to q_i is a valid trace. Let j be the largest number such that the extension of this trace from q_i up to $\langle I, M \rangle = \langle I_i \cup \{b_{i,1}, \ldots b_{i,j}\}, M_i \rangle$ is a valid trace. We analyse the possible values of j, showing that each leads to a contradiction.

— Suppose $0 \leq j < n_i$. First, note that $b_{i,j+1} \notin I \cup M$, because $b_{i,j+1}$ does not occur in $\{y \mid \exists x \succeq_{po} y : x \in \bigcup_{1 \leq \ell \leq i} \alpha_\ell\}$. We need to show that $b_{i,j+1} \in \mathit{Issuable}(\langle I, M \rangle)$. Let $a \in \mathcal{C}$.
  If $a \prec_{po} b_{i,j+1}$, then by the definition of the sequence $b_{i,1}, \ldots b_{i,n_i}$ the element a has been issued already.
  If $a \prec_{mo} b_{i,j+1}$ and a is blocking, then by the definition of $\prec_{mo}$ we have $a \prec_{mo} c$ for $c \in \alpha_{i+1}$ such that $b_{i,j+1} \prec_{po} c$. We show that $a \in \alpha_k$ for some $k < i+1$, proving that $a \in M$:
  — (Case $a, c \in \bar{\mathcal{C}}$.) By applying (12) and (24), possibly multiple times, we establish that $clk_a < clk_c$. Further, for any x and y we have that if r_x or m_x is true and $y \prec_{mo} x$, then m_y is true; this follows from constraints (18) and (20). Hence, m_a is true, and so a must be contained in some α_k, and by a. above we have $k < i+1$.
  — (Case $a \in \bar{\mathcal{C}}$ and $c \in y$ for some $y \in \mathcal{R}$.) Similarly to the above we establish that $clk_a < clk^-_y$, and that because r_y is true, m_a must also hold true. Then by b. above we have that a is contained in some α_k for $k < i+1$.

  - —(Case $a \in y$ for some $y \in \mathcal{R}$ and $b_{i,j+1} \in \bar{\mathcal{C}}$.) Similarly to the above we establish that $clk^+_y < clk_a$, and that because r_y is true, m_a must also hold true. Then by c. above we have that a is contained in some α_k for $k < i + 1$.
  - —(Case $a \in y$ for some $y \in \mathcal{R}$ and $b_{i,j+1} \in z$ for some $z \in \mathcal{R}$ for $y \neq z$.) Here we show that $clk^+_y < clk^-_z$, and then use d. above to prove that a is contained in some α_k for $k < i + 1$.
  - —(Case $a, b \in y$ for some $y \in \mathcal{R}$.) By the definition of the sequence θ and by the definition of the order $\leq_y$ on the elements of y, we have that there is $k < i + 1$ with $a \in \alpha_k$.

- —Suppose $j = n_i$. We have argued above that for every element $b \in \alpha_{i+1}$ and every $a \prec_{mo} b$ we have $a \in M$. Also, $b \in I \setminus M$, and so α_{i+1} is ready in $\langle I, M\rangle$. Finally, we defined α_{i+1} to be either a singleton set containing a wait, or a set containing compatible **send** and **receive**, hence, $\alpha_{i+1} \in \mathit{Matchable}(\langle I, M\rangle)$.

Finally, we argue that the trace is deadlocking. By (15), (16), (19) and the construction of the sequence θ we have that $M_K \subsetneq \mathcal{C}$. We show that from $q_K = \langle I_K, M_K\rangle$ it is not possible to make a match transition, even after possibly making a number of issue transitions. This proves that there is a deadlocking trace. Suppose that it is possible to make a match transition, and let us fix a suffix $q_K \xrightarrow{\{b_1\}} \hat{q}_1 \xrightarrow{\{b_2\}} \hat{q}_2 \ldots \xrightarrow{\{b_n\}} \hat{q}_n \xrightarrow{\alpha} \bar{q}$ where all transitions except for the last one are issue transitions. Note that because $\hat{q}_n = \langle I_K \cup \{b_1, \ldots, b_n\}, M_K\rangle$, for the transition under α to exist it must be the case that for any $b \in \alpha$ and any $a \prec_{mo} b$ we have $a \in M_K$. But then by (18) all $b \in \alpha$ satisfy that r_b is true, and by (17) we get that there is $b \in \alpha$ for which m_b is true, and so $b \in M_K$, which contradicts that the match transition under α can be taken in $\hat{q}_n$. □

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

5.1. Experimental Setup

The MOPPER deadlock discovery tool takes as input a single-path MPI program and outputs the result of the deadlock analysis. Currently, the single-path property is not checked by MOPPER but is assumed to hold.[3] MOPPER is capable in handling blocking and nonblocking point-to-point communication calls and barrier call from the set of collective communication constructs provided by the MPI standard.

MOPPER first compiles and executes the input program using ISP (In-Situ Partial order), presented by Vakkalanka [2010]. The ISP tool outputs a canonical trace of the input program, along with the *matches-before* partial order $\preceq_{mo}$. MOPPER then computes the $\mathbb{M}^+$ over-approximation as follows. The initial $\mathbb{M}^+$ is obtained by taking the union of all sets whose elements are type-compatible (i.e., singleton sets containing a wait call, sets of barrier calls containing individual calls from each process, and sets containing $S_{i,-}(j)$ together with $R_{j,-}(i/*)$), and then refining the set by removing the sets which violate some basic rules implied by $\preceq_{mo}$. Formally, the $\mathbb{M}^+$ we use is the

[3]Note that it is possible to implement an automated heuristic check (at compile time) whether a program conforms to single-path or not. One can begin by performing a context-insensitive analysis of first identifying the control locations where nondeterministic wildcard **receive** calls are posted. Then in a flow-sensitive manner, one can analyze whether the received data from the discovered set of control locations flow in to the predicate of an if-then-else statement that contains a communication call in its body. We leave the development of this aspect of the tool for future.

largest set satisfying

$$\begin{aligned}\mathbb{M}^+ = \{\{a,b\} \mid\ & a = S_{i,-}(j), b = R_{j,-}(i/*),\\ & \forall a' \prec_{mo} a\ \exists\, b' \not\prec_{mo} b : \{a',b'\} \in \mathbb{M}^+,\\ & \forall b' \prec_{mo} b\ \exists\, a' \not\prec_{mo} a : \{a',b'\} \in \mathbb{M}^+\}\\ \cup\ & \{\{a\} \mid a = W_{i,l}(h_j)\}\\ \cup\ & \{\{a_1,\ldots,a_N\} \mid \forall i \in [1,N], a_i = B_{i,-}\}\ .\end{aligned}$$

Due to these conservative constraints, we easily get $\mathbb{M}^+ \supseteq \mathbb{M}$. The definition of the set $\mathbb{M}^+_{\mathcal{R}}$ using $\mathbb{M}^+$ is straightforward.

Example 5.1. To reinforce the intuition behind the set $\mathbb{M}^+$, consider the example given in the following diagram:

P_1	P_2	P_3
$b_1 : bR_{1,1}(*)$	$a_1 : bS_{2,1}(1)$	$a_2 : bS_{3,1}(1)$
$b_2 : bR_{1,2}(*)$		$a_3 : bR_{3,2}(1)$
$b_3 : bS_{1,3}(3)$		$a_4 : bS_{3,3}(1)$
$b_4 : bR_{1,4}(*)$		

Following the definition of $\mathbb{M}^+$, we compute:

$$\mathbb{M}^+ = \{\{a_1,b_1\},\{a_1,b_2\},\{a_1,b_4\},\{a_2,b_1\},\{a_2,b_2\},\{a_3,b_3\},\{a_4,b_4\}\}$$

Notice, however, some matches are infeasible. Under zero-buffering, there does not exist a valid trace of the program where $\{a_1,b_4\}$ is feasible. This discussion illustrates $\mathbb{M}$ might not be equal to $\mathbb{M}^+$.

The partial order $\preceq_{mo}$ and the set $\mathbb{M}^+$ (resp. $\mathbb{M}^+_{\mathcal{R}}$) is then used by MOPPER to construct the propositional formula as explained in the previous section. This propositional formula is passed to the SAT solver, and when the computation finishes, the result is presented to the user, possibly with a deadlocking trace. MOPPER produces not only a SAT model (in case there exists a deadlocking schedule) but also provides an option to the user to re-run the program under the ISP scheduler where the real deadlocking trace is produced by taking cues from the SAT model. Figure 9 illustrates the output of MOPPER on the example code from Figure 2.

Our experiments were performed on a 64-bit, octa-core, 3 GHz Xeon machine with 48 GB of memory, running Linux kernel version 3.19. A time-out of one hour is reserved for ISP, while for MOPPER a time-out of 20 minutes is reserved. MOPPER uses ISP version 0.2.0 [Vakkalanka 2010] to generate the trace and MiniSat version 2.2.0 [Eén and Sörensson 2003] to solve the propositional formula. We used *Totalizer encoding* [Bailleux and Boufkhad 2003] to encode the cardinality constraints that appear in the formulas in Figure 8. All our benchmarks are C MPI programs and the sources of the benchmarks and the MOPPER tool can be found at http://www.github.com/subodhvsharma/benchmarks.git and http://www.github.com/subodhvsharma/mopper-spo.git, respectively.

5.2. Benchmarks

The benchmarks Diffusion2d and Integrate are a part of the FEVS benchmark suite by Siegel and Zirkel [2011a]; these benchmarks exhibit high degree of nondeterminism, as indicated by their value of ρ (see below). The Diffusion2d benchmark solves the two-dimensional diffusion equation. In Diffusion2d, each node communicates its local computation results with its neighbouring nodes which are laid out in a grid fashion. The Integrate benchmark estimates the integral of a sine or a cosine function in a given

```
********* SAT VALUATIONS ************
Number of Clauses: 215
Number of Variables: 78
Constraint Generation Time: 0.000213sec
Solving Time: 0.000128sec
Mem (MB): 4.78516
Formula is SAT -- DEADLOCK DETECTED
M_00:1
I_00:1
M_01:0
I_01:1
M_10:0
I_10:1
M_20:1
I_20:1
S_1000:0
S_2001:0
S_2000:1
============= Running the Program with SAT model =============
Transition list for 0
0 o=8 i=0 rank=0 Recv dl1.c:19 src=-1 rtag=1{[0, 1]} {} Matched [2,0]
1 o=13 i=1 rank=0 Recv dl1.c:20 src=2 rtag=1{} {}

Transition list for 1
0 o=9 i=0 rank=1 Send dl1.c:23 dest=0 stag=1{[1, 1]} {}
1 o=11 i=1 rank=1 Finalize dl1.c:29{} {}
No resource leaks detected, 2 MPI calls trapped.

Transition list for 2
0 o=10 i=0 rank=2 Send dl1.c:26 dest=0 stag=1{[2, 1]} {} Matched [0,0]
1 o=12 i=1 rank=2 Finalize dl1.c:29{} {}
No resource leaks detected, 2 MPI calls trapped.

No matching MPI call found!
Detected a DEADLOCK in interleaving 2
```

Fig. 9. Output of MOPPER on a deadlocking program from Figure 2

range. The integration tasks are dynamically allotted to worker nodes by a master node. Due to this dynamic load balancing by the master node, Integrate is not a single-path MPI program. In order to make Integrate a single path benchmark, we modified the source to implement static load balancing. In this single-path variant of the Integrate benchmark, the schedule space grows as $n!/n$ where n is the number of processes.

The benchmarks Floyd and Gauss Elimination are from Xue et al. [2009] and both are single-path MPI programs. Floyd implements the all-pairs shortest path algorithm and employs a pipelined communication pattern where each process communicates with the process immediately next in the ranking.

We have a set of ten synthetic benchmarks with various deadlocking patterns that are not discovered by the MPI runtime even after repeated runs. Among them, we include only the DTG benchmark (dependence transition group, from Vakkalanka [2010]). The benchmark has a seemingly unrelated pair of matches at the start state that do not commute. Thus, selecting one match-pair over the other leads to a deadlock. A run of ISP with optimization fails to discover the deadlock. However, when the optimization is turned off, ISP discovers the deadlock after three runs.

A pattern similar to the one in DTG exists in the Heat-errors benchmark [Mueller et al. 2011]. This benchmark implements the solution of the heat conduction equation. ISP discovers the deadlock (when this benchmark is run on eight processes) in just over two hours after exploring 5021 interleavings. The same deadlock is discovered in under a second by MOPPER.

5.3. Results

5.3.1. Comparison with ISP. We first compare the performance of MOPPER with the dynamic verifier that is integrated in ISP. The results of the experiments are tabulated in Table I. The table presents the results under different buffering assumptions only for those benchmarks where buffering had an impact.

All the benchmarks have been run 5 times and in the tables the average time taken across all these runs have been reported. The time values reported for MOPPER include the time to (i) generate the execution, (ii) generate the constraints and (iii) solve the constraints. Comparison of the execution time of both tools is meaningful only when the benchmarks are single-path. For the benchmarks where this is not the case MOPPER only explores a subset of the scenarios that ISP explores. To estimate the degree of match nondeterminism in the collected program trace, we introduce a new metric $\rho = |\mathbb{M}^{+}|/mcount$, where $mcount$ is the number of **send** and **receive** matches in the trace. Benchmarks with a high value of ρ have a large set of potential matches. Since the metric relies on potential matches, ρ could be greater than 1 even for a completely deterministic benchmark. From the results it is evident that SAT encoding-based solution outperforms the explicit-state dynamic model checking solution.

5.3.2. Optimized Encoding for Multi-Receives. Next, we compare MOPPER with the optimized encoding for multi-receives presented in Section 4.3. We refer to the novel optimized encoding by MOPPER-Opt. We benchmarked the optimized encoding on those benchmarks where the nondeterminism factor was very high. Incidentally, all these benchmarks also had a high presence of multi-receives. Table II presents a performance analysis of MOPPER-Opt against MOPPER. Note that the optimized encoding produces a dramatic reduction in the analysis time in comparison to the timing results from the original MOPPER encoding. This can be attributed to significant reduction in the size of the formula that gets generated in the optimized encoding which can be seen in Table II. On benchmarks with lower degree of nondeterminism, such as DTG and Gauss Elimination, the performance of both the encodings is similar. Our experience shows that the choice of the cardinality encoding has impact on solving times, with the Totalizer encoding giving the best results.

5.3.3. Comparison with CIVL. We compared MOPPER with a bounded model checker that comes with the CIVL framework [Zirkel et al. 2014; Siegel et al. 2015], version 1.7.

CIVL failed to compile Floyd, Integrate and Diffusion2d. It seems that CIVL does not model many intrinsic functions, which results in compilation failure. For Floyd and Diffusion2d, removing file-related functions such as *printf, fprintf* makes the compilation succeed. However, CIVL crashes with a *NullPointerException* on Diffusion2d in a later phase. For Integrate, we had to add `#define` statements to the primary source file to set values for some constants that are defined in an included header file. This results in a successful compilation of the benchmark.

CIVL crashed with *NullPointerException* on the Heat benchmarks. For the synthesized benchmarks as well as the Integrate benchmarks, we observed that MOPPER was faster by an order of magnitude, as shown in Table III. However, this would not be an objective comparision owing to the way in which CIVL models buffer size. CIVL assumes complete non-determinism in the buffer size; in addition to this, it also considers the scenario in which the buffer size changes in a non-deterministic fashion. Clearly, this makes the set of behaviours explored by CIVL larger than those explored by MOPPER. Furthermore, the analysis performed by CIVL is more static in nature as compared to trace-based analysis of MOPPER. It is worth noting that due to this,

Table I. Experimental Results

						MOPPER			ISP	
B'mark	Calls	Procs	ρ	B	Dl [a]	Vars	Clauses	time	Runs	time
[s] DTG[†]	16	5	1.33	0	✔	270	755	0.04	3	0.17
				∞		256	708	0.06	3	0.29
[s] Gauss Elim	92	8	1.86	0		3.1K	9.6K	0.33	1	1.11
	188	16	1.93	0		7.1K	22.2K	1.54	1	2.40
	380	32	1.97	0		16K	50K	3.07	1	4.00
[s] Heat	152	8	1.8	0	✔	9.4K	28.8K	1.5	>2.5K	TO
	312	16	1.84	0	✔	20.9K	64.6K	4.34	>2.5K	TO
	632	32	1.86	0	✔	47.1K	145.6K	6.98	>2.5K	TO
[s] Floyd	120	8	7	∞		16.3K	60.3K	3.32	>20K	TO
	256	16	7.53	0		41K	152.4K	20.83	>20K	TO
				∞		40.5K	150.9K	31.47	>20K	TO
	528	32	7.8	0		92.4K	344.4K	86.22	>20K	TO
				∞		91.3K	340.9K	155.54	>20K	TO
[s] Diffusion2d	52	4	2.82	∞		3.8K	13.3K	0.39	90	55.76
	108	8	5.7	∞		19K	71.8K	TO	>10.5K	TO
[m] Integrate	28	4	3.0	$\otimes$		386	1163	0.07	6	1.08
	36	8	4.0	$\otimes$		2K	7.3K	0.36	5040	216.72
	46	10	5.0	$\otimes$		3.8K	14.2K	17.34	>13K	TO
	76	16	7.0	$\otimes$		13.8K	53K	TO	>13K	TO

Legend: [a] Deadlock present [†] ISP misses the deadlock under optimized run [TO] Exceeds 20 minute time limit [s] single-path $\otimes$ Buffering model irrelevant [m] modified to be single-path

Table II. Results for MOPPER-Opt encoding

				MOPPER			MOPPER-Opt		
B'mark	Calls	Procs	B	Vars	Cl	time (in sec)	Vars	Cl	time (in sec)
DTG	16	5	0	270	755	0.04	252	695	0.02
			∞	256	708	0.06	238	648	0.06
Gauss Elim	92	8	0	3.1K	9.6K	0.33	3.4K	10.5K	0.32
	188	16	0	7.1K	22.2K	1.54	8K	24.9K	1.64
	380	32	0	16K	50K	3.07	18.3K	57.1K	2.87
[a] Heat	312	16	0	20.9K	64.6K	4.34	19.3K	58K	1.76
	632	32	0	47.1K	145.6K	6.98	42.8K	128.4K	3.63
	1272	64	0	106.2K	331.3K	14.70	93.5K	280.3K	6.89
Floyd	120	8	∞	16.3K	60.3K	3.32	9.6K	30.7K	0.61
	256	16	0	41K	152.4K	20.83	19.7K	62.8K	1.07
			∞	40.5K	150.9K	31.47	19.3K	61.3K	1.68
	528	32	0	92.4K	344.4K	86.22	41.2K	131.2K	2.35
			∞	91.3K	340.9K	155.54	40.1K	127.7K	2.24
	1072	64	0	199.9K	745.7K	856.79	87.9K	279.8K	4.04
			∞	197.6K	737.9K	461.65	85.5K	271.9K	4.19
Diffusion2d	52	4	∞	3.8K	13.3K	0.39	2.5K	7.6K	0.35
	108	8	∞	19K	71.8K	TO	5.1K	15.6K	0.72
	220	16	∞	119.7K	471.4K	TO	10.7K	32.7K	2.02
Integrate	46	10	∞	3.8K	14.2K	17.34	1.1K	3.2K	0.37
	76	16	∞	13.8K	53K	TO	2.1K	6.3K	1.18

Legend: [a] Deadlock present [TO] Exceeds time limit of 20 minutes

Table III. CIVL vs. Simgrid vs. MOPPER-Opt vs. MOPPER

				CIVL	Simgrid	MOPPER-Opt	MOPPER
B'mark	Calls	Procs	B	T (in sec)	T (in sec)	T (in sec)	T (in sec)
[a] Heat	312	16	0	NE	1.44	1.76	4.34
	632	32	0	NE	1.74	3.63	6.98
	1272	64	0	NE	2.31	6.89	14.70
Floyd	120	8	∞	TO	TO	0.61	3.32
Diffusion2d	220	16	∞	NE	[b] 1.32	2.02	TO
Integrate	16	4	∞	[b] 4.62	1.08	0.06	0.07
	26	6	∞	[b] 5.76	1.12	0.06	0.09
	36	8	∞	[b] 7.19	1.21	0.28	0.36
	76	16	∞	[b] 14.37	1.23	1.18	TO

Legend: [a] Deadlock present [b] Wrong result [TO] Exceeds time limit of 20 minutes [NE] NullPointerException

CIVL reports a potential deadlock in the Gaussian Elimination and Integrate benchmarks whereas MOPPER does not. For both of these benchmarks, output of *certainty* parameter of CIVL was *MAYBE*, indicating the uncertainty of the tool regarding the reported deadlocks. The output generated by CIVL is unreadable and insufficient for us to determine whether the cause of deadlock given in the output is spurious. However, through manual inspection and automated inspection of benchmarks using MOPPER, ISP and Simgrid, we conclude that the deadlocks reported by CIVL must be a false positive.

5.3.4. Comparison with TASS. For comparison of MOPPER with bounded model checker TASS by Siegel and Zirkel [2011b], we used the 64-bit Linux binary of TASS version 1.1. Since TASS accepts only a limited subset of C, our experimentation with TASS is restricted to only few benchmarks, namely Integrate and the synthetic benchmarks. With these few benchmarks, the scalability of TASS cannot be evaluated in an objective manner. We observed, however, that the deadlock discovery of TASS on our benchmarks was particularly slow: the analysis of Integrate with TASS timed out when run for ten processes (TASS is configured to time-out after 20 minutes). On the synthetic benchmarks, TASS was one order of magnitude slower than MOPPER.

5.3.5. Comparison with Simgrid. Simgrid, by Merz et al. [2011], is a stateless explicit-state model checker that is integrated into a simulation framework. It is closest to ISP in terms of operationality and methodology. For the evaluation we used real ("wall clock") time for both Simgrid and MOPPER-Opt. On Diffusion2d, Simgrid produces a wrong diagnosis, while on benchmark Floyd, it runs out of time. From our personal conversation with the authors, it became apparent that Simgrid does not analyze `MPI_Send` calls in MPI applications with system buffering. Diffusion2d is known to deadlock in the very first run with zero-buffering. For other benchmarks, such as Integrate, our encoding resulted in significantly smaller analysis time. The results of the comparison of Simgrid with MOPPER-Opt and MOPPER are given in Table III. Note that on the Heat benchmarks, Simgrid performs consistently better than MOPPER but performs similarly to MOPPER-Opt.

Our results show that the search for deadlocks using SAT and our partial-order encoding is efficient compared to existing state-of-the-art dynamic and symbolic verifiers. There is further room for improvement. For example, as operations before a barrier are never matched with operations after the barrier, we can further refine $\mathbb{M}^+$. This could result in a smaller search space for the SAT solver, thereby reducing the constraint solving time.

6. RELATED WORK

Deadlock discovery is a central problem in the CCS community. As an instance, DELFIN$^+$ [Gradara et al. 2006] is a model checker for CCS that uses the A* algorithm as a heuristic to find errors early in the search. Process algebra systems, like CCS and CSP, appear to be a natural fit to analyse MPI programs. However, to the best of our knowledge, no research exists that addresses the problem of automatically building CSP/CCS models from MPI programs and analysing them using CSP/CCS tools. Tools such as Pilot [Carter et al. 2010] support the implementation of CSP models using MPI.

Petri nets are another popular formalism for modelling and analysing distributed systems. Cheng et al. [1995] presented a technique to discover deadlocks in a class of Petri nets called 1-safe Petri nets and proved the problem to be NP-complete. Nevertheless, we are not aware of any polynomial-time reduction between this problem and the problem we study.

Chen et al. [2008] and Wang et al. [2009] apply a predictive trace analysis methodology to multithreaded C/Java programs. Wang et al. [2009] construct a propositional encoding of constraints over partial orders and pass it to a SAT solver. They utilize the source code and an execution trace to discover a causal model of the system that is more relaxed than the causal order computed in some of the prior work in that area. This allows them to reason about a wider set of thread interleavings and find races and assertion violations which other work may miss. The symbolic causal order together with a bound on the number of context switches is used to improve the scalability of the algorithm. In our work, the concept of context switch is irrelevant. The per-process matches-before relation suffices to capture all match possibilities precisely, and consequently, there are neither false positives nor false negatives. The tool presented by Alglave et al. [2013] addresses shared-variable concurrent programs, and is implemented on top of the CBMC Bounded Model Checker by Clarke et al. [2004].

MCAPI (Multicore Communications API) [Holt et al. 2009] is a lightweight message passing library for heterogeneous multicore platforms. It provides support for a subset of the calls found in MPI. For instance, MCAPI does not have deterministic receives or collective operations. Thus, the class of deadlocks found in MCAPI is a subset of the class of deadlocks in MPI. Deniz et al. [2012] provide a trace analysis algorithm that discovers deadlocks and violations of temporal assertions in MCAPI. The discovery of deadlocks is based on the construction of AND Wait-for graphs and is imprecise. The work in [Huang et al. 2013; Elwakil and Yang 2010] discovers assertion violations in MCAPI programs. While both present an order-based encoding, the work by Elwakil and Yang [2010] does not exploit the potential matches relation, and thus yields a much slower encoding, as observed by Huang et al. [2013]. Huang et al. [2013] furthermore present an order-based SMT encoding using the potential matches relation. The encoding is designed to reason about violations of assertions on data, and does not allow to express the existence of deadlocks. The paper furthermore shows that the problem of discovering assertion violations on a trace is NP-complete. Due to the inherent difference of the problems studied, our proof of NP-completeness is significantly more involved than the one of Huang et al. [2013]. In particular, for a 3-CNF formula with n clauses, their work uses n assertions, where each assertion itself is a disjunction of propositions (corresponding to the literals in a clause of the 3-CNF formula). In our case, the satisfiability of all clauses needs to be expressed by a possibility to form a single match.

TASS [Siegel and Zirkel 2011b] is a bounded model checker that uses symbolic execution to verify safety properties in MPI programs that are implemented using a strict subset of C. It is predominantly useful in establishing the equivalence of sequential and parallel versions of a numerically-insensitive scientific computing program. TASS may report false alarms and the authors indicate that the deadlock discovery strategy does not scale when nondeterministic wildcard receives are used [Siegel and Zirkel 2011b].

CIVL is a tool that can analyse C programs that use a variety of communication protocols, including MPI. For verification it uses symbolic execution. Siegel et al. [2015] state that the main strength of CIVL is in the variety of dialects it supports, and that it is outperformed by TASS in their experiments.

Eslamimehr and Palsberg [2014] present a path-based concolic execution engine for discovering deadlocks in Java programs. Key to the technique is the permutation of instructions on the path that is analysed (i.e., the search considers an alternative interleaving). Our SAT-based encoding could be used to simultaneously check an exponential number of the possible permutations.

Fu et al. [2014] (also Fu et al. [2015]) combine an MPI scheduler with a path-based symbolic execution engine, which is based on Cloud9 [Bucur et al. 2011], which in turn is based on KLEE [Cadar et al. 2008]. The symbolic execution engine deals with data nondeterminism, whereas the scheduler explores the communication nondeterminism, in the style of ISP. Nonblocking MPI operations, which yield a large degree of nondeterminism, are not considered. They report experimental results on benchmarks with around six MPI calls on average. The usage of ISP is sufficient to find all deadlocks in those benchmarks that are input independent. Their implementation was not available for comparison on our benchmarks.

López et al. [2015] use a type-based approach for annotating C programs that use the MPI API and supports a broad variety of communication primitives. However, nonblocking operations and wildcard receives are not considered. The type-based approach avoids the analysis of the state space of the MPI program, and thus, does not suffer from the state-space explosion problem. Similarly, Santos et al. [2015] begin with an abstract specification of a protocol that governs the communication between the MPI processes. The verification of the protocols relies on manual annotations. Neither nonblocking calls nor wildcard receives are considered.

Huang and Mercer [2015] present a necessary condition for deadlocks for the case of the zero-buffering model. Zero-buffer encoding discovers a subset of deadlocks discovered by the MOPPER (or MOPPER-Opt) encoding. They report that the performance of their encoding is comparable to ours (for the case of zero-buffer incompatibility) with the exception of the *Diffusion2D* benchmark, where MOPPER times out with 8 and 16 processes. MOPPER-Opt, on the other hand, tackles *Diffusion2D* scalably: for 8 and 16 processes under infinite buffering the analysis requires only 0.02 and 0.04 seconds, respectively. Since their tool for the zero-buffer encoding is not publicly available, a proper comparison with MOPPER-Opt was not feasible. However, the timing results reported by Huang and Mercer [2015] for *Diffusion2D* on 8 and 16 processes were slower than timing results observed with MOPPER-Opt.

Böhm et al. [2016] have recently proposed an explicit-state model checking approach to detect deadlocks in MPI programs. They rely on state pruning by means of partial order reduction (POR) [**?**; **?**]. Their tool supports buffering modes beyond zero and infinite buffering. It remains to be analyzed how this compares to a SAT-based technique, and whether POR could be used as a means to reduce the size of $\mathbb{M}^+$.

7. CONCLUSION

We have investigated the problem of deadlock discovery for a class of MPI programs with no control-flow nondeterminism. We have shown that finding a deadlock in such programs is NP-complete. We have further devised SAT-based encodings that can be successfully used to find deadlocks in real-world programs. We have implemented the encodings as part of a new tool, called MOPPER, and have provided an evaluation on benchmarks of various sizes. Our experiments show that the tool outperforms the state-of-the-art model checkers in the area.

There are several directions in which our tool can be improved. One obvious direction is to handle larger subset of the MPI language. Collective communication calls (with non-synchronizing behaviors) along with a complex interaction of processes in the presence of groups, communicators, and various virtual topologies pose an interesting challenge in extending MOPPER's SAT encoding. Calls such as Wait-some and Wait-any introduce another layer of nondeterminism in the program which may result in much larger state-spaces. Encoding the semantics of such calls in the SAT formulae is rather straightforward. In the recently released MPI 3.0 standard, support for nonblocking collective calls and channel-based communication is proposed. This will require nontrivial changes to our SAT encoding rules, since the semantics of primitives such as nonblocking collectives will be vastly different from the semantics of collective calls in previous versions of the standard. Another interesting direction to take is by making the SAT encoding more efficient; this can be achieved by identifying and breaking symmetries in the SAT formulae. It is long known that symmetry breaking leads to significant reduction in SAT solving times.

While our approach can still be used as a per-path-oracle in a dynamic verifier or model checker that explores the relevant control-flow paths for programs that are not single-path, another important direction will be to make the MOPPER analysis sensitive to the data communicated among processes. This will result in MOPPER alone to handle programs which do not have the single-path property, such as load-balancers, and allow the verification of much larger class of real-world MPI programs. Finally, we believe that the extended encoding for multipath MPI programs will result in wider applicability of our techniques to other popular programming languages that provide message passing support, such as Erlang or Scala. We plan to investigate these in our future work.

Acknowledgements. The authors were in part supported by EPSRC H017585/1, J012564/1 and M023656/1, the EU FP7 STREP PINCETTE, ERC CPROVER 280053 and the H2020 FET OPEN 712689 SC^2. G. Narayanaswamy is a Commonwealth Scholar, funded by the UK government. V. Forejt is also affiliated with Masaryk University, Czech Republic.

REFERENCES

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification (CAV) (LNCS)*, Vol. 8044. Springer, 141–157. DOI:http://dx.doi.org/10.1007/978-3-642-39799-8_9

Olivier Bailleux and Yacine Boufkhad. 2003. Efficient CNF Encoding of Boolean Cardinality Constraints. In *Principles and Practice of Constraint Programming (CP) (LNCS)*, Vol. 2833. Springer, 108–122. DOI:http://dx.doi.org/10.1007/978-3-540-45193-8_8

Stanislav Böhm, Ondrej Meca, and Petr Jancar. 2016. State-Space Reduction of Non-deterministically Synchronizing Systems Applicable to Deadlock Detection in MPI. In *Formal Methods (FM) (LNCS)*, Vol. 9995. 102–118. DOI:http://dx.doi.org/10.1007/978-3-319-48989-6_7

Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *Computer Systems (EuroSys)*. ACM, 183–198. DOI:http://dx.doi.org/10.1145/1966445.1966463

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Operating Systems Design and Implementation (OSDI)*. USENIX Association, 209–224.

John D. Carter, William B. Gardner, and Gary Grewal. 2010. The Pilot Library for Novice MPI Programmers. In *Principles and Practice of Parallel Programming (PPoPP)*. ACM, 351–352. DOI:http://dx.doi.org/10.1145/1693453.1693509

Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. 2008. jPredictor: A predictive runtime analysis tool for Java. In *International Conference on Software Engineering (ICSE)*. ACM, 221–230. DOI:http://dx.doi.org/10.1145/1368088.1368119

Allan Cheng, Javier Esparza, and Jens Palsberg. 1995. Complexity Results for 1-Safe Nets. *Theor. Comput. Sci.* 147, 1&2 (1995), 117–136. DOI:http://dx.doi.org/10.1016/0304-3975(94)00231-7

Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS)*, Vol. 2988. Springer, 168–176. DOI:http://dx.doi.org/10.1007/978-3-540-24730-2_15

Etem Deniz, Alper Sen, and Jim Holt. 2012. Verification and coverage of message passing multicore applications. *ACM Trans. Design Autom. Electr. Syst.* 17, 3 (2012), 23. DOI:http://dx.doi.org/10.1145/2209291.2209296

Niklas Eén and Niklas Sörensson. 2003. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT) (LNCS)*, Vol. 2919. Springer, 502–518. DOI:http://dx.doi.org/10.1007/978-3-540-24605-3_37

Mohamed Elwakil and Zijiang Yang. 2010. Debugging support tool for MCAPI applications. In *Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*. ACM, 20–25. DOI:http://dx.doi.org/10.1145/1866210.1866212

Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: scalable deadlock detection for concurrent programs. In *Foundations of Software Engineering (FSE)*. ACM, 353–365. DOI:http://dx.doi.org/10.1145/2635868.2635918

Vojtech Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2014. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. In *Formal Methods (FM) (LNCS)*, Vol. 8442. Springer, 263–278. DOI:http://dx.doi.org/10.1007/978-3-319-06410-9_19

Xianjin Fu, Zhenbang Chen, Chun Huang, Wei Dong, and Ji Wang. 2014. Synchronization Error Detection of MPI Programs by Symbolic Execution. In *Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 127–134. DOI:http://dx.doi.org/10.1109/APSEC.2014.28

Xianjin Fu, Zhenbang Chen, Yufeng Zhang, Chun Huang, Wei Dong, and Ji Wang. 2015. MPISE: Symbolic Execution of MPI Programs. In *High Assurance Systems Engineering (HASE)*. IEEE, 181–188. DOI:http://dx.doi.org/10.1109/HASE.2015.35

Sara Gradara, Antonella Santone, and Maria Luisa Villani. 2006. DELFIN$^+$: An efficient deadlock detection tool for CCS processes. *J. Comput. Syst. Sci.* 72, 8 (2006), 1397–1412. DOI:http://dx.doi.org/10.1016/j.jcss.2006.03.003

Waqar Haque. 2006. Concurrent deadlock detection in parallel programs. *International Journal in Computer Applications* 28, 1 (2006), 19–25.

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 *High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM. DOI:http://dx.doi.org/10.3233/SPR-130368

Jim Holt, Anant Agarwal, Sven Brehmer, Max Domeika, Patrick Griffin, and Frank Schirrmeister. 2009. Software Standards for the Multicore Era. *IEEE Micro* 29, 3 (2009), 40–51. DOI:http://dx.doi.org/10.1109/MM.2009.48

Yu Huang and Eric Mercer. 2015. Detecting MPI Zero Buffer Incompatibility by SMT Encoding. In *NASA Formal Methods (NFM) (LNCS)*, Vol. 9058. Springer, 219–233. DOI:http://dx.doi.org/10.1007/978-3-319-17524-9_16

Yu Huang, Eric Mercer, and Jay McCarthy. 2013. Proving MCAPI executions are correct using SMT. In *Automated Software Engineering (ASE)*. IEEE, 26–36. DOI:http://dx.doi.org/10.1109/ASE.2013.6693063

Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. 2003. MARMOT: An MPI Analysis and Checking Tool. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications (PARCO)* (2005-02-07) *(Advances in Parallel Computing)*. Elsevier, 493–500. DOI:http://dx.doi.org/10.1016/S0927-5452(04)80063-7

Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. 2012. Verifying GPU kernels by test amplification. In *PLDI*. ACM, 383–394. DOI:http://dx.doi.org/10.1145/2254064.2254110

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*. ACM, 280–298. DOI:http://dx.doi.org/10.1145/2814270.2814302

Glenn R. Luecke, Yan Zou, James Coyle, Jim Hoekstra, and Marina Kraeva. 2002. Deadlock detection in MPI programs. *Concurrency and Computation: Practice and Experience* 14, 11 (2002), 911–932. DOI:http://dx.doi.org/10.1002/cpe.701

Stephan Merz, Martin Quinson, and Cristian Rosa. 2011. SimGrid MC: Verification Support for a multi-API Simulation Platform. In *FMOODS/FORTE (LNCS)*, Vol. 6722. Springer, 274–288. DOI:http://dx.doi.org/10.1007/978-3-642-21461-5_18

Message Passing Interface Forum. 2009. MPI: A Message-Passing Interface Standard. (2009). http://www.mpi-forum.org/docs/mpi-2.2.

Matthias S. Mueller, Ganesh Gopalakrishnan, Bronis R. de Supinski, David Lecomber, and Tobias Hilbrich. 2011. Dealing with MPI Bugs at Scale: Best Practices, Automatic Detection, Debugging, and Formal Verification. (2011). http://rcswww.zih.tu-dresden.de/~hilbrich/sc11/.

N. Natarajan. 1984. A distributed algorithm for detecting communication deadlocks. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS) (LNCS)*, Vol. 181. Springer, 119–135. DOI:http://dx.doi.org/10.1007/3-540-13883-8_68

César Santos, Francisco Martins, and Vasco Thudichum Vasconcelos. 2015. Deductive Verification of Parallel Programs Using Why3. In *ICE*. DOI:http://dx.doi.org/10.4204/EPTCS.189.11

Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. 2009. MCC: A runtime verification tool for MCAPI user applications. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 41–44. DOI:http://dx.doi.org/10.1109/FMCAD.2009.5351145

Stephen F. Siegel. 2007. Model checking nonblocking MPI programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS)*, Vol. 4349. Springer, 44–58. DOI:http://dx.doi.org/10.1007/978-3-540-69738-1_3

Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Jackie Kern and Jeffrey S. Vetter (Eds.). ACM, 61:1–61:12. DOI:http://dx.doi.org/10.1145/2807591.2807635

Stephen F. Siegel and Timothy K. Zirkel. 2011a. FEVS: A Functional Equivalence Verification Suite for High-Performance Scientific Computing. *Mathematics in Computer Science* 5, 4 (2011), 427–435. DOI:http://dx.doi.org/10.1007/s11786-011-0101-6

Stephen F. Siegel and Timothy K. Zirkel. 2011b. *The Toolkit for Accurate Scientific Software*. Technical Report UDEL-CIS-2011/01. Department of Computer and Information Sciences, University of Delaware.

Sarvani Vakkalanka. 2010. *Efficient dynamic verification algorithms for MPI applications*. Ph.D. Dissertation. University of Utah.

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 *Computer Aided Verification (CAV) (LNCS)*, Vol. 5123. Springer, 66–79. DOI:http://dx.doi.org/10.1007/978-3-540-70545-1_9

Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE. http://dx.doi.org/10.1109/SC.2010.7

Chao Wang, Sudipta Kundu, Malay K. Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *FM (LNCS)*, Vol. 5850. Springer. DOI:http://dx.doi.org/10.1007/978-3-642-05089-3_17

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. In *PPoPP*. ACM. DOI:http://dx.doi.org/10.1145/1504176.1504213

Timothy K. Zirkel, Stephen F. Siegel, and Louis F. Rossi. 2014. *Using Symbolic Execution to Verify the Order of Accuracy of Numerical Approximations*. Technical Report UD-CIS-2014/002. Department of Computer and Information Sciences, University of Delaware.