2016-05-01

An Analyzer for Message Passing Programs

Yu Huang
Brigham Young University - Provo

An Analyzer for Message Passing Programs

Yu Huang

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Eric G. Mercer, Chair
Michael D. Jones
Jay A. McCarthy
William Arthur Barrett
Tony R. Martinez

Department of Computer Science

Brigham Young University

May 2016

ABSTRACT

An Analyzer for Message Passing Programs

Yu Huang
Department of Computer Science, BYU
Doctor of Philosophy

Asynchronous message passing systems are fast becoming a common means for communication between devices. Two problems existing in message passing programs are difficult to solve. The first problem, intended or otherwise, is message-race where a receive may match with more than one send in the runtime system. This non-determinism often leads to intermittent and unexpected behavior depending on the resolution of the race. Another problem is deadlock, which is a situation in that 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. Detecting if message-race and/or deadlocks exist in a message passing program are both NP–complete. The difficulty of solving the two problems also comes from three factors that complicate the semantics: asynchronous communication, synchronous barrier, and buffering settings including infinite buffering (the system can buffer messages) and zero buffering (the system has no internal buffering).

To solve the above problems with complicating factors, this research provides a novel predictive analysis that initializes a concrete execution and then predicts the behavior of other executions that arise from the initial execution. This research starts with Satisfiability Modulo Theories (SMT) based model checking that provides precise analysis for the program behavior. Unfortunately, a precise analysis using SMT does not scale to large programs. As such, the SMT based model checking is combined with heuristic search for witnessing program properties. The heuristic search is efficient in identifying how sends may match with receives in the runtime as it only looks for the match relations for sends and receives in a small searching space initially; the space is increased only if the program property is not witnessed, until all possible match relations for sends and receives reflected in message non-determinism are found. This research also gives a static analysis approach that is scalable as it does not need to analyze the full set of program behaviors; rather, the static analysis only uses polynomial-time algorithms to identify all potential deadlocks in a send-receive templates given a set of pre-defined deadlock patterns. Given the predictive analysis consisting of SMT based model checking with heuristic search and static analysis, this research is able to solve the two problems above. The work in this dissertation also demonstrates that the predictive analysis is more efficient than the existing tools for verifying message passing programs.

Keywords: Message passing, MPI, MCAPI, SMT, Static analysis, Model checking

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my advisor, Eric Mercer, who has continually given me persistent support and excellent guidance in every step of my research and study. His enthusiastic and meticulous attitude to research greatly inspired me and guided me to become a qualified researcher. Without his help, completing this dissertation would not be possible.

I would also like to thank Jay McCarthy, who worked closely with Eric and me in the early years of this research. He gave me a lot of interesting and excellent ideas and suggestions for both the current and future work of this research. I also thank Mike Jones, Willam Barrett and Tony Martinez, who gave me important suggestions that strengthened the research.

I also appreciate the help from all the present and graduated students in our lab, specially Saint Wesonga, Neha Rungta, Mark O'Neill, Eric Noonan, Peter Anderson, Benjamin Hillery and Josh Asplund.

Last but not least, I am grateful for the strongest encouragement and support from my sweet family during the years. I am proud of them!

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statements and Importance

Nowadays, asynchronous message passing is widely used in communication between devices such as medical devices, network infrastructures and automobiles. High performance computing (HPC) also needs message passing to communicate among multiple cores.

The essential structure of asynchronous message passing is the use of two communication primitives, send and receive, for message communication. Figure 1.1 shows a simple example of an asynchronous message passing program with two processes running in parallel. The program uses only blocking sends (s) and blocking receives (r). A send or receive specifies the "from" and the "to" process IDs. Given the scenario in Figure 1.1, the message passing is accomplished by matching s_0 and r_0 in the runtime.

With the increasing usage of message passing, numerous issues leading to intermittent and unexpected results become essential to message passing. The primary source of intermittent or unexpected program behavior is message-race. A message-race is generally a concern in any message passing program. It is defined as a situation where a receive may match with more than one send in the runtime system. As a result, the send matched with

p_0	p_1
r_0(from p_1, to p_0)	s_0(from p_1, to p_0)

Figure 1.1: A simple asynchronous message passing program.

the receive is non-deterministic as it depends on the runtime system and other external influences. The problem of determining if a message-race exists in a message passing program is NP–complete. The message race may lead to invalid computation by the program or even deadlock. Such non-determinism, although sometimes intended, makes debug and test difficult because they need to explore the complete program state space which cannot be trivially accomplished.

Another problem in message passing programs is deadlock. There are two ways to deadlock: first, message sends and receives resolve into a mutual dependency; and second, a message is delayed from ever being received. The work in this dissertation assumes weak fairness meaning that no process will be isolated indefinitely so every issued send will be received eventually, and as such, deadlock can only arises from a mutual dependency. The problem of verifying the existence of deadlock from mutual dependency is also NP–complete. This type of deadlock must be checked as message-race or unexpected communication pattern that can lead to it.

1.2 Complicating Factors

There are three aspects that complicates the message passing semantics: asynchronous communication, synchronous barriers and buffering settings. The asynchronous communication is accomplished by a set of primitives such as send, receive or barrier, where they may match in the runtime to transmit messages. The complexity of asynchronous communication is that the order of these primitives can be resolved in many ways. For example, for a send and a receive that may match in the runtime, either the send or the receive can be issued first, where the match occurs.

The barriers are able to synchronize a program among a group of processes in such a way that each group member waits at a position in a single process until all the members are witnessed. The barrier synchronization may impact asynchronous message passing as the messages are not blocked by synchronization; instead, they can migrate across barriers.

This is because a send in asynchronous communication may be issued before the barriers witness and is delayed to match a receive until the barrier synchronization completes.

The message passing semantics are also complicated because of the two buffering settings: infinite buffering (the system has internal buffering for the messages) and zero buffering (the system has no internal buffering). According to the two buffering semantics, the program behavior may be different. For example, a program under zero buffer semantics may deadlock while a feasible schedule without deadlock may exist for the same program under infinite buffer semantics.

In summary, the dissertation focuses on solving the problems of message-race and deadlock under two buffering settings. To be precise, the question of message-race asks if there is an incorrect computation as a result of message-race where the messages are received in such a way that the program output is incorrect. The question of deadlock asks if there is a schedule where two or more operations each are waiting for the other to finish, and thus neither ever does. Note that the deadlock may occur via message-race.

To understand the complexity of message passing and begin working on potential solutions, this dissertation looks at two common message passing standards: Multicore Association Communications API (MCAPI) [37] and Message Passing Interface (MPI) [3]. This research starts with a precise SMT based model checking on MCAPI, where a simple point-to-point communication that only supports non-deterministic receives are specified. This research then expands and scales the precise SMT encoding to MPI, where a more complicated point-to-point communication with both non-deterministic and deterministic receives and the barrier synchronization are defined.

1.3 Related Works

The primary challenge of any approach to detecting deadlock or message-race is scalability. The scalability is difficult to accomplish because of the large number of commands in message passing programs and complicated program semantics.

Model checking with SMT provides precise analysis. Model checking technique generates a model for the input program, and then checks program properties in the model. The SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Model checking with SMT leverages SMT to encode program behavior such as program ordering, message communication, and valid computation as constraints in an SMT problem. Existing SMT tools, e.g., Yices [2] and Z3 [13], are then used to solve this problem by resolving these constraints. The prior SMT techniques are not scalable as they encode too many formulas for the program behavior, and therefore do not complete within a short time because of the state explosion [17, 18]. Forejt et al. proposed a SAT based approach to detect deadlock in single-path MPI programs [20]. The solution is correct and efficient for programs with a low degree of message non-determinism. However, since the size of the encoding is cubic, checking large programs is time consuming.

The dynamic analysis is performed by executing a program. The existing dynamic analysis based model checkers are also inefficient to reason about the program behavior as they exhaustively enumerate all possible program interleavings which can be exponential [48, 49, 57, 59]. The first push button model checker of MCAPI applications, MCC, systematically explores all relevant process interleavings using the Dynamic Partial Order Reduction (DPOR) algorithm [48]. This model checker uses match pair – a coupling of a send and a receive that may potentially match in the runtime, to capture the non-determinism of an MCAPI program. The POE algorithm also uses DPOR [19] in the verifier, ISP, for verifying MPI programs [57, 59]. This algorithm operates by postponing the cooperative operations for message passing in transit until each process reaches a blocking call. It then determines the potential matches of send and receive operations in the runtime. An extension to the POE algorithm is the MSPOE [49] that is able to detect communication deadlock in MPI programs.

There are many other works that use different approaches in message passing verification. Umpire is an approach of runtime verification for checking multiple MPI errors such as deadlock and resource tracking [60]. The error checking is taken by spawning one manager thread and several outfielder threads in the execution of an MPI program. A drawback of the approach is that it relies on a concrete execution, which may miss the errors in the other execution trace. An extension to Umpire is Marmot [32]. The work uses a centralized sever instead of multiple threads for error checking. Another extension to Umpire is MUST [23]. The structure of MUST allows users to execute the error checking either in an application process itself or in extra processes that are used to offload these analyses. However, just like Umpire and Marmot, the approach is neither sound nor complete for deadlock detection.

MPIDD, like Umpire, has a central manager that traps all MPI calls using the MPI profiling interface; however Umpire runs as a separate process and communicates using shared memory with different processes [22]. MPIDD runs as another MPI process and the trapped information is sent to the central detector using MPI calls. MPIDD is essentially a deadlock detector. It creates a dependency graph to figure out the potential/real deadlocks. The detection algorithm is a depth first search for cycles in the dependency graph.

MPI-CHECK uses a macro-like mechanism wherein the MPI calls in the program are instrumented to have extra arguments [35]. These arguments provide information such as line number in the source code where the call was made, the MPI function name and its arguments. The information is recorded in a database known as the Program Database (PDB). The process of checking is split into two phases. In the first phase, the instrumentation of MPI programs is performed followed by their compilation. In the second phase, the execution of the instrumented MPI code under the control of the MPI-CHECK server takes place.

MPIRace-Check is a tool that identifies message-race [43]. MPIRace-Check uses vector clocks to discover the racing sends. MPIRace-Check does not have the ability to

deterministically replay the program. Since vector clocks are used, MPIRace-Check has scalability issues.

Vo. et al. proposed a dynamic verification approach for large-scale MPI programs using lamport clocks with lazy updates [62, 63]. The auxiliary information by setting up a logic clock is sent out and updated via piggyback messages. This method detects the potential matches of send and receive operations with respect to the piggyback data assigned to each operation. This method scales well for MPI applications with large numbers of processes and messages. However, this method misses potential matches of sends and receives, therefore making the state exploration incomplete.

MPI-Spin is integrated in the classic model checker – SPIN [24], for verifying MPI programs [52]. It generates a model of an MPI program and allows one to symbolically execute it. However, it fails to model non-terminating execution sequences because of the nature of symbolic execution.

Bouajjani et al. proposed an analysis technology of message passing programs that bounds the number of process communication cycles [5]. This bounding scheme reduces concurrency to a non-deterministic sequential program. However, this method restricts the range of the pending message such that the "task" is the only format. Also, this method can only resolve message non-determinism caused by "post" operations. Further, this method has no way to handle the collective operations. As such, this analysis can not be applied for the paradigm that uses sends and receives for message passing. Synchronous primitives such as the barriers in MPI programs hinder the program behavior analysis. Some works aim to ease the analysis by detecting the semantic relations among synchronous primitives.

One of the works is an algorithm that detects the matching of textually unaligned barriers - the common synchronization primitive in an SPMD–style (single program, multiple data) program, i.e. programs that are allowed in the MPI or OpenMP specifications [66]. Instead of checking a barrier matching, another work is able to detect the irrelevant barriers

[47]. The work relies on the definitions of IntraCB and InterCB that imply the "complete before" relations among processes.

The Threaded-C Bounded Model Checking (TCBMC) extends the C Bounded Model Checking (CBMC) [10, 11] to support concurrent C program verification [44]. The approach bounds the number of context switching allowed among threads because it assumes that most bug patterns have only a few context switchings. Especially, the work assumes there are no nested lock-unlock patterns.

Burckhardt et al. presented the CheckFence prototype [7] that exhaustively checks all executions of a test program by translating a program into SAT formulas. It increments the observations each time by adding more constraints to SAT formulas.

Dubrovin et al. give a method to translate an asynchronous system into a set of transition formulas over three partial order semantics [15]. The encoding adds constraints to compress the search space and decreases the bound on program unwinding.

SATCheck is a scalable technique for model checking concurrent programs based on concrete execution [14]. The approach first builds a model of an observed execution in an SAT encoding, then gets an assignment of the encoding that represents an unobserved execution, builds a model of this execution and repeats the above steps until all the executions are observed. As such, all the reachable behaviors are explored.

1.4 Our Solution

Solving the problems of message-race and deadlock with both precision and scalability are difficult. This dissertation looks to predictive analysis that detects program properties in a set of executions that arise from an initial concrete execution. This dissertation defines a precise encoding that leverages advances in SMT technology, and then scales the encoding using heuristic search and static analysis.

The SMT encoding in this dissertation directly uses send-receive match pair to capture how many ways to resolve message non-determinism. It also precisely constrains the program

order as well as the buffering semantics into a set of formulas. As such, if an SMT solver resolves a violating schedule, the violation actually exists in the program.

The SMT based model checking does not scale well as the runtime cost is highly dependent on how many ways to resolve message non-determinism. As such, the work in this dissertation combines the precise SMT encoding with heuristic search that is able to under-approximate the resolutions of message non-determinism. Initially, the heuristic search approximates a subset of the precise match pairs that are encoded to the SMT problem for property checking. A property may be a message-race or a deadlock. If a property is detected with the initial search, then the verification completes; otherwise, the approach iterates to a larger set of program behaviors and repeats to detect properties until the full set of the program behaviors is reached. The property detection can be more efficient in an early iteration of the heuristic search.

Another scalable approach in this dissertation is static analysis. The static analysis does not rely on the full set of program behaviors; rather, it approximates the existence of properties by statically searching an abstraction of the program. This static searching relies on algorithms with low cost of runtime. The feasibility of the detected properties relies on further validation such as the SMT based model checking. If no property is detected, however, the program is free of that property.

1.5 Thesis Statement

Verifying message-race and deadlock in message passing programs are NP–complete problems that can be checked and scaled by predictive analysis including precise SMT encoding with heuristic search and static analysis.

1.6 Our Contributions

This section presents an overview of the various research contributions of this dissertation. Each contribution is an important element in verifying message-race and deadlock using the work in this dissertation. We list each paper published for a research contribution and a discussion on the research contribution.

1.6.1 Proving MCAPI Executions Are Correct using SMT

- Yu Huang, Eric Mercer, Jay McCarthy. "Proving MCAPI Executions are Correct Applying SMT Technology to Message Passing". In Proceeding of IEEE/ACM International Conference on Automated Software Engineering. Palo Alto, CA, November, 2013, 26 - 36.
- *Abstract.* Chapter 2 provides a way to encode an MCAPI execution as a SMT problem, which if satisfiable, yields a feasible execution schedule on the same trace, such that it resolves non-determinism in the MCAPI runtime in a way that it now fails user provided assertions. The work proves the problem is NP-complete. The encoding is useful for test, debug, and verification of MCAPI program execution. The novelty in the encoding is the direct use of match pairs (potential send and receive couplings). Match-pair encoding for MCAPI executions, when compared to other encoding strategies, is simpler to reason about, results in significantly fewer terms in the SMT problem, and captures feasible behaviors that are ignored in previously published techniques. Further, to our knowledge, this is the first SMT encoding that is able to run in infinite-buffer semantics, meaning the runtime has unlimited internal buffering as opposed to no internal buffering. Results demonstrate that the SMT encoding, restricted to zero-buffer semantics, uses fewer clauses when compared to another zero-buffer technique, and it runs faster and uses less memory. As a result the encoding scales well for

programs with high levels of non-determinism in how sends and receives may potentially match.

- *Contribution to thesis.* The work in Chapter 2 establishes the complexity of the problem of verifying message-race and gives the SMT encoding for precise analysis. The solution is able to verify the correctness of message passing executions.

1.6.2 Detecting MPI Zero Buffer Incompatibility by SMT Encoding

- Yu Huang, Eric Mercer. "Detecting MPI Zero Buffer Incompatibility by SMT Encoding". In Proceeding of the 7th NASA Formal Methods Symposium. Pasadena, CA, April, 2015.

- *Abstract.* Chapter 3 presents an algorithm that encodes a single-path MPI program as a SMT problem, which if satisfiable, yields a feasible schedule, such that it proves the program is zero buffer compatible. This encoding is also adaptable to checking assertion violation for correct computation. To support MPI semantics, this algorithm correctly defines the point-to-point communication and collective communication with respective rules for both infinite buffer semantics and zero buffer semantics. The novelty in this work is considering only the schedules that strictly alternate sends and receives leading to an intuitive zero buffer encoding. This work proves that the set of all the strictly alternating schedules is capable of covering all the message communication that may occur in any execution under zero buffer semantics. Experiments demonstrate that the SMT encoding is correct and highly efficient for a set of benchmarks compared with two state-of-the-art MPI verifiers.

- *Contribution to thesis.* The work in Chapter 3 gives an extension to the SMT encoding in Chapter 2 that supports zero buffer semantics and collective operations in MPI semantics. The encoding is capable of detecting zero buffer incompatibility, which is a type of deadlock under zero buffer semantics. The solution is more efficient than two state-of-art MPI verifiers, ISP [19] and MOPPER [20].

1.6.3 An Efficient Approach for Match Pair Approximation in Message Passing

- Yu Huang, Eric Mercer. "An Efficient Approach for Match Pair Approximation in Message Passing". Submitted to 8th NASA Formal Methods Symposium. Minneapolis, MN, June, 2016.
- *Abstract.* Chapter 4 presents a new algorithm that under-approximates the match pairs iteratively: first sectioning each process in the CTP such that each potential sender distributes roughly a bounded number of sends to match the same number of receives in the process, and then approximating the match pairs for the sends and receives in each section independently based on the match generation algorithm in Chapter 2. The algorithm runs in quadratic complexity in the number of operations. Novel in the work is that the algorithm has the flexibility to generate the match pair set with various size based on the user input. This work further shows that the precise match pairs for any CTP can be generated with a bounded input. The experiments over a set of benchmarks show that the algorithm in this work drastically reduces the runtime performance of property witnessing as all the properties are witnessed with a small set of match pairs generated by the new algorithm. The results also show that the algorithm is able to scale to a program that employs a high degree of message non-determinism and/or a high degree of deep communication.
- *Contribution to thesis.* The work in Chapter 4 is a heuristic search that is able to reduce the size of the match set as input to an SMT encoding for verifying a message passing program. The work is helpful for large programs to be feasible as the over-approximated match set for such programs is usually too large to be resolved.

1.6.4 A Hybrid Approach of Dynamic and Static Analyses for Deadlock in MPI Programs

- Yu Huang, Eric Mercer. "A Hybrid Approach of Dynamic and Static Analyses for Deadlock in MPI Programs". Submitted to The International Symposium on Software Testing and Analysis. Saarbrcken, Germany, July, 2016.
- *Abstract.* Chapter 5 addresses the NP–complete problem of deadlock detection in MPI programs in the context of a concurrent trace program (CTP). The initial trace program is preprocessed by executing the program once. The solution uses progressively more precise analyses to generate and then prune a potential set of deadlocks: static matching to identify deadlock pattern instances; execution of the instances on an abstract machine to reject those that are provably non-feasible; and finally, if needed, validation of the instances to remove any remaining that are non-feasible. Novel in the work is the abstract machine based on counting to efficiently reject many non-feasible instances without exhaustively enumerating all message races. The work further defines two deadlock patterns: circular dependency and orphaned receive. The first pattern relies on simple rules for validation, while the second requires a higher cost SMT encoding. The work proves the approach sound and complete for each pattern and compares the approach with two other deadlock tools on typical benchmarks. The comparison shows that the new approach scales in the presence of millions of possible send-receive matches and completes on benchmarks where the other tools time out. The experiments also show that the two deadlock patterns in the work cover all the deadlock cases in benchmarks.
- *Contribution to thesis.* The work in Chapter 5 combines static analysis and precise SMT technique for detecting deadlocks in MPI programs. The solution is more efficient than two state-of-the-art MPI verifiers, ISP [19] and MOPPER [20].

1.7 Summary

This dissertation gives a solution that is able to verify message-race and deadlock in message passing programs for both infinite buffer and zero buffer semantics. The solution uses model checking with SMT for precise analysis, and is further scaled with heuristic search and static analysis. The solution is applied to two prevalent message passing standards, MCAPI and MPI. The work in this dissertation is more efficient than the other existing tools including other SMT/SAT encoding techniques and dynamic analyses. As such, the work in this dissertation is able to scale to large message passing programs.

Chapter 2

Proving MCAPI Executions Are Correct using SMT

Abstract

Asynchronous message passing is an important paradigm in writing applications for embedded heterogeneous multicore systems. The Multicore Association (MCA), an industry consortium promoting multicore technology, is working to standardize message passing into a single API, MCAPI, for bare metal implementation and portability across platforms. Correctness in such an API is difficult to reason about manually, and testing against reference solutions is equally difficult as reference solutions implement an unknown set of allowed behaviors, and programmers have no way to directly control API internals to expose or reproduce errors. This paper provides a way to encode an MCAPI execution as a Satisfiability Modulo Theories (SMT) problem, which if satisfiable, yields a feasible execution schedule on the same trace, such that it resolves non-determinism in the MCAPI runtime in a way that it now fails user provided assertions. The paper proves the problem is NP-complete. The encoding is useful for test, debug, and verification of MCAPI program execution. The novelty in the encoding is the direct use of match pairs (potential send and receive couplings). Match-pair encoding for MCAPI executions, when compared to other encoding strategies, is simpler to reason about, results in significantly fewer terms in the SMT problem, and captures feasible behaviors that are ignored in previously published techniques. Further, to our knowledge, this is the first SMT encoding that is able to run in *infinite-buffer* semantics, meaning the runtime has unlimited internal buffering as opposed to no internal

buffering. Results demonstrate that the SMT encoding, restricted to zero-buffer semantics, uses fewer clauses when compared to another zero-buffer technique, and it runs faster and uses less memory. As a result the encoding scales well for programs with high levels of non-determinism in how sends and receives may potentially match.

2.1 Introduction

Embedded devices fill all sorts of crucial roles in our lives. They exist as medical devices, as network infrastructure, and they control our automobiles. Embedded devices continue to become more powerful as computing hardware becomes smaller and more modular. It is now becoming commonplace to find multiple processing units inside a single device. The Multicore Association (MCA) is an industry group that has formed to define specifications for low-level communication, resource management, and task management for embedded heterogeneous multicore devices [36].

One specification that the MCA has released is the Multicore Association Communications API (MCAPI) [37]. The specification defines types and functions for simple message passing operations between different computing entities within a device. Messages can be passed across persistent channels that force an ordering of the messages, or they can be passed to specific *endpoints* within the system. The specification places few ordering constraints on messages passed from one endpoint to another. This freedom introduces the possibility of a race between multiple messages to common endpoints thus giving rise to non-deterministic behavior in the runtime [42]. If an application has non-determinism, it is not possible to test and debug such an application without a way to directly (or indirectly) control the MCAPI runtime.

There are two ways to implement the MCAPI semantics: infinite-buffer semantics (the message is copied into a runtime buffer on the API call) and zero-buffer semantics (the message has no buffering) [59]. An infinite-buffer semantics provides more non-deterministic behaviors in matching send and receives because the runtime can arbitrarily delay a send to create interesting (and unexpected) send reorderings. The zero-buffer semantics follow intuitive message orderings as a send and receive essentially rendezvous.

Sharma et al. propose a method to indirectly control the MCAPI runtime to verify MCAPI programs under zero-buffer semantics [48]. As the work does not address infinite-buffer semantics, it is somewhat limited in its application. The work does provide a dynamic

partial order reduction for the model checker, but such a reduction is not sufficient to control state space explosion in the presence of even moderate non-determinism between message sends and receives. A key insight from the approach is its direct use of match pairs–couplings for potential sends and receives.

Wang *et al.* propose an alternative method for resolving non-determinism for program verification using symbolic methods in the context of shared memory systems [64]. The work observes a program trace, builds a partial order from that trace called a concurrent trace program (CTP), and then creates an SMT problem from the CTP that if satisfied indicates a property violation.

Elwakil *et al.* extend the work of Wang *et al.* to message passing and claim the encoding supports both infinite and zero buffer semantics. A careful analysis of the encoding, however, shows it to not work under infinite-buffer semantics and to miss behaviors under zero-buffer semantics [17]. Interestingly, the encoding assumes the user provides a precise set of match pairs as input with the program trace, and it then uses those match pairs in a non-obvious way to constrain the happens-before relation in the encoding. The work does not discuss how to generate the match pairs, which is a non-trivial input to manually generate for large or complex program traces. An early proof claims that the problem of finding a precise set of match pairs given a program trace is NP-complete [45].

This paper presents a proof that resolving non-determinism in message passing programs in a way that meets all assertions is NP-complete. The paper then presents an SMT encoding for MCAPI program executions that works for both zero and infinite buffer semantics. The encoding does require an input set of match pairs as in prior work, but unlike prior work, the match-set can be over-approximated and the encoding is still sound and complete. The encoding requires fewer terms to capture all possible program behavior when compared to other proposed methods making it more efficient in the SMT solver. To address the problem of generating match pairs, an algorithm to generate the over-approximated set is given. To summarize, the main contributions in this paper are

1. a proof that the problem of matching sends to receives in a way that meets assertions is NP-complete;
2. a correct and efficient SMT encoding of an MCAPI program execution that detects all program errors under zero or infinite buffer semantics given the input set of potential match pairs contains at least the precise set of match pairs; and
3. an $O(N^2)$ algorithm to generate an over-approximation of possible match pairs, where N is the size of the execution trace in lines of code.

Organization: Section 2 gives an example. Section 3 shows the NP-completeness reduction. Section 4 gives the encoding. Section 5 shows how to generate match-pairs. Section 6 presents the results. Section 7 discusses related work. And Section 8 is conclusions and future work.

2.2 Example

It is a challenge to explain intended behavior in simple scenarios consisting of a handful of calls when dealing with concurrency. Consider the MCAPI program execution in Figure 2.1 that includes three tasks that use send (`mcapi_msg_send_i`) and receive (`mcapi_msg_recv_i`) calls to communicate with each other. Line numbers appear in the first column for each task with the first digit being the task ID. The declarations of the local variables are omitted for space.

Picking up the scenario just after the endpoints are defined, lines `02` and `05` receive two messages on the endpoint *e0* in variables A and B which are converted to integer values and stored in variables a and b on lines `04` and `07`; task 1 receives one message on endpoint *e1* in variable C on line `13` and then sends the message *"1"* on line `15` to *e0*; and finally, task 2 sends messages *"4"* and *"Go"* on lines `24` and `26` to endpoints *e0* and *e1* respectively. The additional code (lines `08 - 09`) asserts properties of the values in a and b. The `mcapi_wait` calls block until the associated send or receive buffer is able to be used. Given the scenario, a

Task 0	Task 1	Task 2
00 initialize(NODE_0,&v,&s);	10 initialize(NODE_1,&v,&s);	20 initialize(NODE_2,&v,&s);
01 e0=create_endpoint(PORT_0,&s);	11 e1=create_endpoint(PORT_1,&s);	21 e2=create_endpoint(PORT_2,&s);
	12 e0=get_endpoint(NODE_0,PORT_0,&s);	22 e0=get_endpoint(NODE_0,PORT_0,&s);
02 msg_recv_i(e0,A,sizeof(A),&h1,&s);		23 e1=get_endpoint(NODE_1,PORT_1,&s);
03 wait(&h1,&size,&s,MCAPI_INF);	13 msg_recv_i(e1,C,sizeof(C),&h3,&s);	
04 a=atoi(A);	14 wait(&h3,&size,&s,MCAPI_INF);	24 msg_send_i(e2,e0,"4",2,N,&h5,&s);
		25 wait(&h5,&size,&s,MCAPI_INF);
05 msg_recv_i(e0,B,sizeof(B),&h2,&s);	15 msg_send_i(e1,e0,"1",2,N,&h4,&s);	
06 wait(&h2,&size,&s,MCAPI_INF);	16 wait(&h4,&size,&s,MCAPI_INF);	26 msg_send_i(e2,e1,"Go",3,N,&h6,&s);
07 b=atoi(B);		27 wait(&h6,&size,&s,MCAPI_INF);
	17 finalize(&s);	
08 if(b > 0);		28 finalize(&s);
09 assert(a == 4);		
0a finalize(&s);		

Figure 2.1: An MCAPI concurrent program execution

developer might ask the question: *"What are the possible values of* `a` *and* `b` *after the scenario completes?"*

The intuitive trace is shown in Figure 2.2 using a shorthand notation for the MCAPI commands: send (denoted as `S`), receive (denoted as `R`), or wait (denoted as `W`). The shorthand notation further preserves the thread ID and line number as follows: for each command

24 $\mathtt{S}_{2,4}(0, \&\mathtt{h5})$
25 $\mathtt{W}(\&\mathtt{h5})$
---
02 $\mathtt{R}_{0,2}(2, \&\mathtt{h1})$
03 $\mathtt{W}(\&\mathtt{h1})$
---
26 $\mathtt{S}_{2,6}(1, \&\mathtt{h6})$
27 $\mathtt{W}(\&\mathtt{h6})$
---
04 $\mathtt{a} = \mathtt{atoi}(\mathtt{A});$
---
13 $\mathtt{R}_{1,3}(2, \&\mathtt{h3})$
14 $\mathtt{W}(\&\mathtt{h3})$
15 $\mathtt{S}_{1,5}(0, \&\mathtt{h4})$
16 $\mathtt{W}(\&\mathtt{h4})$
---
05 $\mathtt{R}_{0,5}(1, \&\mathtt{h2})$
06 $\mathtt{W}(\&\mathtt{h2})$
07 $\mathtt{b} = \mathtt{atoi}(\mathtt{B});$
08 $\mathtt{assume}(\mathtt{b} > 0);$
09 $\mathtt{assert}(\mathtt{a} == 4);$

Figure 2.2: A feasible execution traces of the MCAPI program execution in Figure 2.1

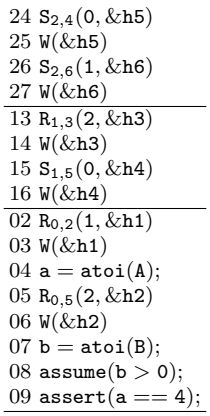

Figure 2.3: A second feasible execution traces of the MCAPI program in Figure 2.1

$\mathtt{O}_{\mathtt{i},\mathtt{j}}(\mathtt{k}, \&\mathtt{h})$, $\mathtt{O} \in \{\mathtt{S}, \mathtt{R}\}$ or $\mathtt{W}(\&\mathtt{h})$, $\mathtt{i}$ represents the task ID, $\mathtt{j}$ represents the source line number, $\mathtt{k}$ represents the destination endpoint, and $\mathtt{h}$ represents the command handler. A specific destination task ID is in the notation when a trace is fully resolved, otherwise "*" indicates that a receive has yet to be matched to a send. The lines in the trace indicate the context switch where a new task executes.

From the trace, variable a should contain 4 and variable b should contain 1 since task 2 must first send message *"4"* to *e0* before it can send message *"Go"* to *e1*; consequently, task 1 is then able to send message *"1"* to *e0*. The assume notation asserts the control flow taken by the program execution. In this example, the program takes the true branch of the condition on line 08. At the end of execution the assertion on line 09 holds and no error is found.

There is another feasible trace shown in Figure 2.3 which is reachable under the infinite-buffer semantics. In this trace, the variable a contains 1 instead of 4, since the message *"1"* is sent to *e0* after sending the message *"Go"* to *e1* as it is possible for the send on line 24 to be buffered in transit. The MCAPI specification indicates that the wait on line 25 returns once the buffer is available. That only means the message is somewhere in the runtime and not that the message is delivered. As such, it is possible for the message to be buffered in transit allowing the send on line 15 to arrive at *e0* first and be received in

variable *"a"*. Such a scenario is a program execution that results in an assertion failure at line 09.

From the discussion above, it is important to consider non-determinism in the MCAPI runtime when testing or debugging an MCAPI program execution. The next section presents a proof that the problem of matching sends to receives in a way that meets all assertions is NP-complete. The proof justifies the encoding and SMT solver. Following the proof, the algorithm to generate the encoding is presented. It takes an MCAPI program execution with a set of possible send-receive match pairs and generates an SMT problem that if satisfied proves that non-determinism can be resolved in a way that violates a user provided assertion (the assertions are negated in the encoding) and if unsatisfiable proves the trace correct (meaning the user assertions hold on the given execution under all possible runtime behaviors). The encoding can be solved by an SMT solver such as Yices [16] or Z3 [13].

2.3 NP Completeness Proof

The complexity proof is inspired by the NP-completeness proof for memory coherence and consistency by Cantin *et al.* that uses a similar reduction from SAT only in the context of shared memory [9]. The complexity proof is on a new decision problem: *Verifying Assertions in Message Passing* (VAMP).

Definition 1. ***Verifying Assertions in Message Passing.***

INSTANCE: A set of constants D, a set of variables X, and a finite set H of task histories consisting of send, receive, and assert operations over X and D.

QUESTION: Is there a feasible schedule S for the operations of H that satisfy all the assertions?

The VAMP problem is NP-complete. The proof is a reduction from SAT. Given an instance Q of SAT consisting of a set of variables U and set of clauses C over U, an instance V of VAMP is constructed such that V has a feasible schedule S that meets all the assertions

if and only if there is a satisfying truth assignment for Q. Feasible in this context means the schedule is allowed by the MCAPI semantics.

The reduction is illustrated in Figure 2.4. The figure elides the explicit calls to wait which directly follow each send and receive operation, and it elides the subscript notation as it is redundant in the figure. The figure also adds the value sent and the variable that receives the value to the notation as that information is pertinent to the reduction.

The reduction relies on non-determinism in the message passing to decide the value of each variable in U. The tasks h_{d_0} and h_{d_1} repeatedly send the constant value d_0 (false valuation) or d_1 (true valuation) to task h_C. The key intuition is that these tasks are synchronized with h_C so they essentially wait to send the value until asked.

The task h_C sequentially requests and receives d_0 and d_1 values for each variable in the SAT instance Q. It does not request values for a new variable until the current variable is resolved. As the values come from two separate tasks upon request, the messages race in the runtime and may arrive in either order at h_C. As a result, the value in each variable is non-deterministically d_0 or d_1.

After the value of each variable u_i is resolved, the h_C task asserts the truth of each clause in the problem instance. As the clauses are conjunctive, the assertions are sequentially evaluated. If a satisfying assignment exists for Q, then a feasible schedule exists that resolves the values of each variable in such a way that every assert holds.

Lemma 1. *S is a feasible schedule for H that satisfies all assertions if and only if Q is satisfiable.*

Proof. **Feasible schedule for V implies Q is satisfiable**: proof by contradiction. Assume that Q is unsatisfiable even though there is a feasible schedule S for V that meets all the assertions. The reduction in Figure 2.4 considers all truth values of the variables in Q, over every combination, by virtue of the non-determinism, and then asserts the truth of each of the clauses in Q. The complete set of possibilities is realized by sending in parallel from h_{d_0} and h_{d_1} the two truth valuations for a given variable to h_C. As these messages

SAT: $U \equiv \{u_0, u_1, ..., u_m\}$ $C \equiv \{c_0, c_1, ..., c_n\}$ $Q \equiv \{c_0 \wedge c_1 \wedge ... \wedge c_n\}$		
VAMPI: $H \equiv \{h_{d_0}, h_{d_1}, h_C\}$ $X \equiv \{u_0, ..., u_m, g_0, g_1\}$ $D \equiv \{d_0, d_1\}$		
h_{d_0}	h_{d_1}	h_C
$R(g_0, *)$ $S(d_0, h_C)$	$R(g_1, *)$ $S(d_1, h_C)$	$S(d_0, h_{d_0})$ $S(d_0, h_{d_1})$ $R(u_0, *)$ $R(u_0, *)$
$R(g_0, *)$ $S(d_0, h_C)$	$R(g_1, *)$ $S(d_1, h_C)$	$S(d_0, h_{d_0})$ $S(d_0, h_{d_1})$ $R(u_1, *)$ $R(u_1, *)$
...	...	...
		$assert(c_0)$ $assert(c_1)$ ...

Figure 2.4: General SAT to VAMPI reduction

may be received in any order, each variable may assume either truth value. Further, each variable resolved is an independent choice so all combinations of variable valuations must be considered. This fact is a contradiction to the assumption of Q being unsatisfiable as the same truth values that meet the assertions would be a satisfying assignment in Q.

Q is satisfiable implies feasible schedule for V: the proof is symmetric to the previous case and proceeds in a like manner. □

Theorem 1 (NP-complete)**.** *VAMP is NP-complete.*

Proof. **Membership in NP**: a certificate is a schedule matching send and receives in each of the histories. The schedule is linearly scanned with the histories and checked that it does not violate MCAPI semantics. Our extended version constructs an operational model of MCAPI semantics that does just such a check given a schedule [27]. The complexity is linear in the size of the schedule.

NP-hard: polynomial reduction from SAT. The correctness of the reduction is established by Lemma 1. The remainder of the proof is the complexity of the reduction. There are two tasks to send values d_0 and d_1 upon request. For each variable $u_i \in U$, each of these tasks, d_0 and d_1, needs two operations: one to synchronize with h_C and another to send the value: $O(2*2*|U|)$. The task h_C must request values from h_{d_0} and h_{d_1} and then receive both those values; it must do this for each variable: $O(2*2*|U|)$. Once all the values are collected, it must them assert each clause: $O(|C|)$. As every term is linear, the reduction is linear. □

2.4 SMT Encoding

The new SMT encoding is based on (1) a trace of events during an execution of an MCAPI program including control-flow assumptions and property assertions, such as Figure 2.2; and (2) a set of possible match pairs. A match pair is the coupling of a receive to a particular send. In the running example, the set admits, for example, that $\mathtt{R}_{0,2}$ can be matched with either $\mathtt{S}_{1,5}$ or $\mathtt{S}_{2,4}$. This direct use of match pairs, rather than a state-based or indirect use of match pairs in an order-based encoding, [17] and [18], is novel.

The purpose of the SMT encoding is to force the SMT solver to resolve the match pairs for the system in such a way that the final values of program variables meet the assumptions on control flow but violate some assertion. In essence, the SMT solver completes a partial order on operations into a total order that determines the final match pair relationships.

2.4.1 Definitions

The encoding needs to express the partial order imposed by the MCAPI semantics as SMT constraints. The partial order is based on a *Happens-Before* relation over operations such as send, receive, wait, or assert:

Definition 2 (Happens-Before)**.** *The* Happens-Before (HB) *relation, denoted as* $\prec_{\mathtt{HB}}$*, is a partial order over operations.*

Given two operations, A and B, if A must complete before B in a valid program execution, then $A \prec_{\texttt{HB}} B$ will be an SMT constraint.

The relation is derived from the program source and potential match pairs. In order to specify the constraints from the program source, each program operation is mapped to a set of variables that can be manipulated by the SMT solver.

Definition 3 (Wait)**.** *The occurrence of a wait operation,* `W`*, is captured by a single variable,* $order_{\texttt{W}}$*, that constrains when the wait occurs.*

It is not enough to represent all events as simple numbers that will be ordered in this way. Such an encoding would not allow the solver to discover what values would flow across communication primitives. Instead, some events in the trace are modeled as a set of SMT variables that record the pertinent information about the event. For example,

Definition 4 (Send)**.** *A send operation* `S`*, is a four-tuple of variables:*

1. $M_{\texttt{S}}$*, the order of the matching receive event;*
2. $order_{\texttt{S}}$*, the order of the send;*
3. $e_{\texttt{S}}$*, the endpoint; and,*
4. $value_{\texttt{S}}$*, the transmitted value.*

The endpoints do not change and the transmitted values are constants in an SMT encoding mainly because this static topology has already been evaluated in an existing execution trace once the trace was obtained. The most complex operation in MCAPI is a receive. Since receives are inherently asynchronous, it is not possible to represent them atomically. Instead, we need to associate each receive with a wait that marks where in the program the receive operation is guaranteed to be complete. The MCAPI runtime semantics allow a single wait to witness the completion of many receives due to *the message non-overtaking property.* A wait that witnesses the completion of one or more receives is the *nearest-enclosing wait.*

Task 0	Task 1
$\mathtt{R}_{0,1}(*, \&\mathtt{h1})$	$\mathtt{S}_{1,1}(0, \&\mathtt{h3})$
$\mathtt{R}_{0,2}(*, \&\mathtt{h2})$	$\mathtt{W}(\&\mathtt{h3})$
$\mathtt{W}(\&\mathtt{h2})$	$\mathtt{S}_{1,2}(0, \&\mathtt{h4})$
$\mathtt{W}(\&\mathtt{h1})$	$\mathtt{W}(\&\mathtt{h4})$

Figure 2.5: Nearest-enclosing Wait example

Definition 5 (Nearest-Enclosing Wait)**.** *A wait that witnesses the completion of a receive by indicating that the message is delivered and that all the previous receives in the same task issued earlier are complete as well.*

Figure 2.5 shows that the wait $\mathtt{W}(\&\mathtt{h2})$ witnesses the completion of the receive $\mathtt{R}_{0,1}$ and $\mathtt{R}_{0,2}$ in task 0. Thus, $\mathtt{W}(\&\mathtt{h2})$ is their nearest-enclosing wait.

The encoding requires that every receive operation have a nearest-enclosing wait as it makes match pair decisions at the wait operation. The requirement is not a limitation of the encoding, as accessing a buffer from a receive that does not have a nearest-enclosing wait is an error. Rather, the wait is a convenience in the encoding to mark where a receive actually takes place. The same requirement can be made for sends for correctness but is not required for the encoding as send buffering is handled differently than receive buffering. The encoding effectively ignores wait operations for sends as will be seen.

Definition 6 (Receive)**.** *A receive operation* $\mathtt{R}$ *is modeled by a five-tuple of variables:*

1. $M_{\mathtt{R}}$*, the order of the matching send event;*
2. $order_{\mathtt{R}}$*, the order of the receive;*
3. $e_{\mathtt{R}}$*, the endpoint;*
4. $value_{\mathtt{R}}$*, the received value; and,*
5. $nw_{\mathtt{R}}$*, the order of the nearest enclosing wait.*

2.4.2 Assumptions, Assertions, and match pairs

The definitions so far merely establish the pertinent information about each event in the trace as SMT variables. It is necessary to now express constraints on those variables.

The most trivial kind of constraints are those for control-flow assumptions.

Definition 7 (Assumption)**.** *Every assumption* $\texttt{A}$ *is added as an SMT assertion.*

It may seem strange to turn *assumptions* into *assertions*, but from a constraint perspective, the assumption that we have already observed some property (during control-flow) is equivalent to instructing the SMT solver to treat it as inviolate truth, or an assertion.

The next level of constraint complexity comes from property assertions. These correspond to the invariants of the program. The goal is to discover if they can be violated, so we instruct the SMT solver to seek for a way to satisfy their *negation* given all the other constraints.

Definition 8 (Property Assertion)**.** *For every property assertion* $\texttt{P}$*,* $\neg\texttt{P}$ *is added as an SMT assertion.*

Finally, we must express the relation in a given match pair as a set of SMT constraints. Informally, a match pair equates the shared components of a send and receive and constrains the send to occur before the nearest-enclosing wait of the receive. Formally:

Definition 9 (Match Pair)**.** *A match pair,* $\langle \texttt{R}, \texttt{S} \rangle$*, for a receive* $\texttt{R}$ *and a send* $\texttt{S}$ *corresponds to the constraints:*

1. $M_{\texttt{R}} = order_{\texttt{S}}$
2. $M_{\texttt{S}} = order_{\texttt{R}}$
3. $e_{\texttt{R}} = e_{\texttt{S}}$
4. $value_{\texttt{R}} = value_{\texttt{S}}$ *and*
5. $order_{\texttt{S}} \prec_{\texttt{HB}} nw_{\texttt{R}}$

The encoding is given a set of potential match pairs over all the sends and receives in the program trace. The constraints from these match pairs are not simply joined in a conjunctions. If we were to do that, then we would be constraining the system such that a single receive must be paired with all possible sends in a feasible execution rather than a

single send. Therefore, we combine all the constraints for a given receive with all possible sends as specified by the input match pairs into a single disjunction:

Definition 10 (Receive Matches)**.** *For each receive* $\mathtt{R}$*, if* $\langle \mathtt{R}, \mathtt{S}_0 \rangle$ *through* $\langle \mathtt{R}, \mathtt{S}_n \rangle$ *are match pairs, then* $\bigvee_i^n \langle \mathtt{R}, \mathtt{S}_i \rangle$ *is used as an SMT constraint.*

This encoding of the input ensures that the SMT solver can only use compatible send/receive pairs and ensures that sends happen before nearest-enclosing waits on receives.

2.4.3 Program Order Constraints

The encoding thus far is missing additional constraints on the *Happens-Before* relation stemming from program order. These constraints are added in four steps: we must ensure that sends to common endpoints occur in program order in a single task (step 1); similarly for receives (step 2); receives occur before their nearest-enclosing wait (step 3); and, that sends are received in the order they are sent (step 4).

Step 1 For each task, if there are sequential send operations, say $\mathtt{S}$ and $\mathtt{S}'$, from that task to a common endpoint, $e_{\mathtt{S}} = e_{\mathtt{S}'}$, then those sends must follow program order: $order_{\mathtt{S}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$.

Step 2 For each task, if there are sequential receive operations, say $\mathtt{R}$ and $\mathtt{R}'$, in that task on a common endpoint, $e_{\mathtt{R}} = e_{\mathtt{R}'}$, then those receives must follow program order: $order_{\mathtt{R}} \prec_{\mathtt{HB}} order_{\mathtt{R}'}$.

Step 3 For every receive $\mathtt{R}$ and its nearest enclosing wait $\mathtt{W}$, $order_{\mathtt{R}} \prec_{\mathtt{HB}} order_{\mathtt{W}}$.

Step 4 For any pair of sends $\mathtt{S}$ and $\mathtt{S}'$ on common endpoints, $e_{\mathtt{S}} = e_{\mathtt{S}'}$, such that $order_{\mathtt{S}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$, then those sends must be received in the same order: $M_{\mathtt{S}} \prec_{\mathtt{HB}} M_{\mathtt{S}'}$.

For example, consider two tasks where task 0 sends two messages to task 1 as shown in Figure 2.6. The $M_{\mathtt{S}}$ variables from the sends will be assigned to the orders for $\mathtt{R}_{1,1}$ and

Task 0	Task 1
$\mathtt{S}_{0,1}(1, \&\mathtt{h1})$	$\mathtt{R}_{1,1}(*, \&\mathtt{h3})$
$\mathtt{S}_{0,2}(1, \&\mathtt{h2})$	$\mathtt{R}_{1,2}(*, \&\mathtt{h4})$
$\mathtt{W}(\&\mathtt{h1})$	$\mathtt{W}(\&\mathtt{h3})$
$\mathtt{W}(\&\mathtt{h2})$	$\mathtt{W}(\&\mathtt{h4})$

Figure 2.6: Send Ordering Example

$\mathtt{R}_{1,2}$ by the match pairs selected by the SMT solver. The constraints added in this step force the send to be received in program order using the HB relation which for this example yields $M_{\mathtt{S}_{0,1}} \prec_{\mathtt{HB}} M_{\mathtt{S}_{0,2}}$.

2.4.4 Zero Buffer Semantics

The constraints presented so far correspond to an infinite-buffer semantics, because we do not constrain how many messages may be in transit at once. We can add additional, orthogonal, constraints to further restrict behavior and enforce a zero-buffer semantics. There are two kinds of such constraints.

First, for each task, if there are two sends S and S′ such that $order_{\mathtt{S}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$, and S and S′ can both match a receive R, then we add the following constraint to the encoding: $order_{\mathtt{W}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$ where W is the nearest-enclosing wait that witnesses the completion of R in execution.

The second constraint relies on a dependence relation between two match pairs.

Definition 11. *To match pairs are dependent, denoted as* $\langle \mathtt{R}, \mathtt{S} \rangle \rightharpoonup \langle \mathtt{R}', \mathtt{S}' \rangle$*, if and only if*

1. *the nearest-enclosing wait* W *of* R′ *issues before* S *on an identical endpoint; or*
2. $\exists \langle \mathtt{R}''\mathtt{S}'' \rangle$ *such that* $\langle \mathtt{R}, \mathtt{S} \rangle \rightharpoonup \langle \mathtt{R}'', \mathtt{S}'' \rangle \wedge \langle \mathtt{R}'', \mathtt{S}'' \rangle \rightharpoonup \langle \mathtt{R}', \mathtt{S}' \rangle$.

With the dependence relation, the second set of constraints for the zero-buffer semantics is given as: for each pair of sends S and S′ that can both match a receive R, if there is a send S″ issued after the issuing of S′ by an identical endpoint, and a receive R′ such that $\langle \mathtt{R}, \mathtt{S} \rangle \rightharpoonup \langle \mathtt{R}', \mathtt{S}'' \rangle$, then we add the following constraint to the encoding: $order_{\mathtt{W}} \prec_{\mathtt{HB}} order_{\mathtt{S}}$ where W is the nearest-enclosing wait that witnesses the completion of R.

```
...
01 order_{R_{0,2}} ≺_HB order_{W(&h1)}
02 order_{R_{0,5}} ≺_HB order_{W(&h2)}
03 order_{R_{0,2}} ≺_HB order_{R_{0,5}}
04 order_{R_{1,3}} ≺_HB order_{W(&h3)}
05 (> b 0)
06 (not (= a 4))
07 ⟨R_{0,2},S_{2,4}⟩ ∨ ⟨R_{0,2},S_{1,5}⟩
08 ⟨R_{0,5},S_{2,4}⟩ ∨ ⟨R_{0,5},S_{1,5}⟩
09 ⟨R_{1,3},S_{2,7}⟩
```

Figure 2.7: SMT Encoding

2.4.5 Example

Figure 2.7 shows the encoding of Figure 2.1 as an SMT problem. We elide the basic definition of the variables discussed in Section 2.4.1. Lines 05 through 09 give the assumptions, assertions, and match pairs. The first four lines reflect the program order constraints: receives happen before corresponding wait operations and receives from a common endpoint follow program order. There are no constraints between sends because there are no sequential sends from a common endpoint to a common endpoint. To encode the zero-buffer semantics, the constraint $order_{\texttt{W(\&h1)}} \prec_{\texttt{HB}} order_{\texttt{S}_{1,5}}$ would need to be added to force the receive to complete before another send is issued.

2.4.6 Correctness

Before we can state our correctness theorem, we must define a few terms. We define our encoder as a function from programs and match pair sets to SMT problems:

Definition 12 (Encoder)**.** *For all programs, p, and match pair sets m, let $\mathcal{SMT}(p, m)$ be our encoding as an SMT problem.*

We assume that an SMT solver can be represented as a function that takes a problem and returns a satisfying assignment of variables or an unsatisfiable flag:

Definition 13 (SMT Solver)**.** *For all SMT problems, s, let $\mathcal{SOL}(s)$ be in $\sigma + \mathbb{UNSAT}$, where σ is a satisfying assignment of variables to values.*

We assume that from a satisfying assignment to one of our SMT problems, we can derive an execution trace by observing the values given to each of the $order_e$ variables. In other words, we can view the SMT solver as returning traces and not assignments.

We assume a semantics for traces that gives their behavior as either having an assertion violation or being correct: [1]

Definition 14 (Semantics)**.** *For all programs, p, and traces t, $\mathcal{SEM}(p,t)$ is either $\mathbb{BAD}$ or $\mathbb{OK}$.*

Given this framework, our SMT encoding technique is sound if

Theorem 2 (Soundness)**.** *For all programs, p, and match pair sets, m, $\mathcal{SOL}(\mathcal{SMT}(p,m)) = t \Rightarrow \mathcal{SEM}(p,t) = \mathbb{BAD}$.*

Our soundness proof relies on the following lemma:

Lemma 2. *Any match pair $\langle \mathtt{R}, \mathtt{S} \rangle$ used in a satisfying assignment of an SMT encoding is a valid match pair and reflects an actual possible MCAPI program execution.*

Proof. We prove this by contradiction. First, assume that $\langle \mathtt{R}, \mathtt{S} \rangle$ is an invalid match pair (i.e. one that is not valid in an actual MCAPI execution). Second, assume that the SMT solver finds a satisfying assignment.

Since $\langle \mathtt{R}, \mathtt{S} \rangle$ is not a valid match pair, match `R` and `S` requires program order, message non-overtaking, or no-multiple match to be violated. In other words, the *Happens-Before* constraints encoded in the SMT problem are not satisfied.

[1] In fact, our extended technical report [27] gives such a semantics.

This is a contradiction: either the SMT solver would not return an assignment or the match pair was actually valid. □

The correctness of our technique relies on completeness:

Theorem 3 (Completeness)**.** *For all programs, p, and traces, t, $\mathcal{SEM}(p,t) = \mathbb{BAD} \Rightarrow \exists m.\mathcal{SOL}(\mathcal{SMT}(p,m)) = t$.*

We prove completeness in our extended version [27] by designing our semantics, $\mathcal{SEM}$, such that it simulates the solving of the SMT problem during its operation to ensure that the two make identical conclusions.

However, these theorems obscure an important problem: how do we know which match pair set to use? Soundness assumes we have one, while completeness merely asserts that one exists. Although Section 2.5 discusses our generation algorithm, we prove here an additional theorem that asserts that any conservative over-approximation of match pair sets is safe.

Theorem 4 (Approximation)**.** *Give two match pair sets m and m', $m \subseteq m' \Rightarrow \mathcal{SOL}(\mathcal{SMT}(p,m)) \sqsubseteq \mathcal{SOL}(\mathcal{SMT}(p,m'))$, where $\mathbb{UNSAT} \sqsubseteq \sigma$.*

Informally, this is true because larger match pair sets only allow *more* behavior, which means that the SMT solver has more freedom to find violations, but that all prior violations are still present. However, because of soundness, it is not possible that using a larger match pair set will discover false violations. The formal proof, in our extended version [27], relies on a match set combination operator that we prove distributes over an essential part of the semantics.

2.5 Generating Match Pairs

The exact set of match pairs can be generated by simulating the program trace and using a depth-first search to enumerate non-determinism arising from concurrent sends and receives.

```
// initialization
input an MCAPI program
initialize list_r
initialize list_s

// check each receive and send with the same endpoint
for r in list_r
  for s in list_s
    let dest = destination endpoint(s)
    let src = source endpoint(s)
    // check matching criteria for r and s
    if
      1. endpoint(r) = dest
      2. index(r) >= index(s)
      3. index(r) =< (index(s)
                     + count(sends(dest=dest))
                     - count(sends(src=src, dest=dest)))
    then
      add pair (r, s) to match_set
    else
      continue
    end if
  end for
end for

output match_set;
```

Figure 2.8: Pseudocode for generating over-approximated match pairs

Such an effort, however, solves the entire problem at once because if you simulate the program trace exploring all non-determinism, then you may as well verify all runtime choices for property violations at the same time.

In this section, we present an algorithm that does not require an exhaustive enumeration of runtime behavior in simulation. Our algorithm over-approximates the match pairs such that match pairs that can exist in the runtime are all included and some bogus match pairs that cannot exist in the runtime may or may not be included. This algorithm does well in restricting the size of bogus match pairs where each one is a non-deterministic choice

that costs an SMT solver more runtime and system memory. Generating a precise set of match-pairs is NP-complete [45].

The algorithm generates the over-approximated match pair set by matching each pair of the send and receive commands at common endpoints and then pruning obvious matches that cannot exist in any runtime implementation of the specification.

Figure 2.8 presents the major steps of the algorithm. The algorithm proceeds by first linearly traversing each task of the program storing each receive and send command into two distinct structured lists. The receive list, `list_r`, is structured as in (2.1) and the send list, `list_s`, is structured as in (2.2).

$$\begin{aligned}
&(\mathtt{e}_0 \ \rightarrow \ ((0,\ \mathtt{R}_{0,1}),\ (1,\ \mathtt{R}_{0,2}),\ \ldots)) \\
&(\mathtt{e}_1 \ \rightarrow \ ((0,\ \mathtt{R}_{1,1}),\ (1,\ \mathtt{R}_{1,2}),\ \ldots)) \\
&\ldots \\
&(\mathtt{e}_n \ \rightarrow \ ((0,\ \mathtt{R}_{n,1}),\ (1,\ \mathtt{R}_{n,2}),\ \ldots))
\end{aligned} \tag{2.1}$$

The list `list_r` groups receives by the issuing endpoint. The integer field merely records the order in which the receives are issued and increases by one on each receive. Similarly, the list `list_s` groups sends first by the destination endpoint and then by the source endpoint. Like `list_r`, an index increases by one to track the issue order. As the input is a program execution trace, any sends or receives in loops already have unique identifiers.

$$\begin{aligned}
&\text{“}\textit{dest}\text{”} \quad \text{“}\textit{src}\text{”} \qquad\qquad\qquad\qquad\qquad\qquad \text{“}\textit{src}\text{”} \\
&(\mathtt{e}_0 \ \rightarrow \ ((\mathtt{e}_1 \ \rightarrow \ ((0,\ \mathtt{S}_{1,1}),\ (1,\ \mathtt{S}_{1,2}),\ \ldots),\ (\mathtt{e}_2 \ \rightarrow \ (\ldots), \\
&\ldots)))) \\
&(\mathtt{e}_1 \ \rightarrow \ ((\mathtt{e}_0 \ \rightarrow \ ((0,\ \mathtt{S}_{0,1}),\ (1,\ \mathtt{S}_{0,2}),\ \ldots),\ (\mathtt{e}_2 \ \rightarrow \ (\ldots), \\
&\ldots)))) \\
&\ldots \\
&(\mathtt{e}_n \ \rightarrow \ ((\mathtt{e}_0 \ \rightarrow \ ((0,\ \mathtt{S}_{0,3}),\ (1,\ \mathtt{S}_{0,4}),\ \ldots),\ (\mathtt{e}_1 \ \rightarrow \ (\ldots), \\
&\ldots))))
\end{aligned} \tag{2.2}$$

Consider the program in Figure 2.9. The lists `list_r` and `list_s` for the program are

$$\begin{aligned}
&(0 \ \rightarrow \ ((0,\ \mathtt{R}_{0,1}),\ (1,\ \mathtt{R}_{0,2}),\ (2,\ \mathtt{R}_{0,4}))) \\
&(1 \ \rightarrow \ ((0,\ \mathtt{R}_{1,2})))
\end{aligned} \tag{2.3}$$

Task 0	Task 1	Task 2
$\mathtt{R}_{0,1}(*, \&\mathtt{h1})$	$\mathtt{S}_{1,1}(0, \&\mathtt{h5})$	$\mathtt{S}_{2,1}(0, \&\mathtt{h8})$
$\mathtt{W}(\&\mathtt{h1})$	$\mathtt{W}(\&\mathtt{h5})$	$\mathtt{W}(\&\mathtt{h8})$
$\mathtt{R}_{0,2}(*, \&\mathtt{h2})$	$\mathtt{R}_{1,2}(*, \&\mathtt{h6})$	
$\mathtt{W}(\&\mathtt{h2})$	$\mathtt{W}(\&\mathtt{h6})$	
$\mathtt{S}_{0,3}(1, \&\mathtt{h3})$	$\mathtt{S}_{1,3}(0, \&\mathtt{h7})$	
$\mathtt{W}(\&\mathtt{h3})$	$\mathtt{W}(\&\mathtt{h7})$	
$\mathtt{R}_{0,4}(*, \&\mathtt{h4})$		
$\mathtt{W}(\&\mathtt{h4})$		

Figure 2.9: Another MCAPI concurrent program

$$\begin{array}{l}(0 \rightarrow ((1 \rightarrow ((0,\ \mathtt{S}_{1,1}),\ (1,\ \mathtt{S}_{1,3})),\ (2 \rightarrow ((0,\ \mathtt{S}_{2,1}))))))\\ (1 \rightarrow ((0 \rightarrow ((0,\ \mathtt{S}_{0,3})))))\end{array} \tag{2.4}$$

The sends $\mathtt{S}_{1,1}$, $\mathtt{S}_{1,3}$, and $\mathtt{S}_{2,1}$ have task 0 as an identical destination endpoint. The send $\mathtt{S}_{0,3}$ has task 1 as the destination endpoint. The list `list_s` in (2.4) reflects this partition. Receive $\mathtt{R}_{0,1}$ is the first receive operation in endpoint 0. This fact is again reflected in `list_r` in (2.3).

The algorithm traverses the two lists in a nested loop to generate match pairs between send and receive commands. The function `index(r)` takes the endpoint of the receive and returns the issue order of that receive in the `list_r` structure. Similarly, the function `index(s)` takes the destination and source endpoints in the send and returns the issue order of that send in the `list_s` structure. These indexes help track message non-overtaking.

The criteria to generate a match pair first requires the send and receive to be compatible (check 1), consistent with message non-overtaking (check 2), and that message non-overtaking does not preclude the match (check 3). A match is precluded by message non-overtaking when a receive cannot possibly match a send because by the time the program issues the receive, the send must have already been matched somewhere else. The function `count` gives the number of sends to a specific destination or the number of sends to a specific source and destination. As long as a receive is issued early enough to still match the send given the message non-overtaking rule, then the match is possible.

In our concrete example, $\mathtt{R}_{0,1}$ is matched with $\mathtt{S}_{1,1}$ or $\mathtt{S}_{2,1}$, but it cannot be matched with $\mathtt{S}_{1,3}$ since the second rule is not satisfied such that the order of $\mathtt{R}_{0,1}$ is less than the order of $\mathtt{S}_{1,3}$ (i.e., $\mathtt{S}_{1,3}$ would have to overtake $\mathtt{S}_{1,1}$ to satisfy the rule). The match between $\mathtt{R}_{0,4}$ and $\mathtt{S}_{1,1}$ is also precluded by check 3 as $\mathtt{S}_{1,1}$ must have already matched an earlier receive by message non-overtaking.

The generated set of match pairs for our example in Figure 2.9 is over-approximated by the algorithm because it includes pairs that cannot exist in any feasible execution. For example, the match pair ($\mathtt{S}_{2,1}$ $\mathtt{R}_{0,4}$) is not feasible because it is not possible to order $\mathtt{S}_{1,3}$ before $\mathtt{R}_{0,2}$ since $\mathtt{R}_{1,2}$ can only match with $\mathtt{S}_{0,3}$ that must occur after $\mathtt{R}_{0,2}$. Fortunately, a satisfying solution is only possible using feasible match pairs. Non-feasible match pairs merely result in extra clauses in the encoding and potentially slow down the SMT solver.

The complexity of the algorithm is quadratic. Traversing the tasks to initialize the lists is $O(N)$, where N is the total lines of code of the program. Traversing the list of receives and the list of sends takes $O(mn)$ to complete, where m is the total number of sends and n is the total number of receives. As $m+n \leq N$, the algorithm takes $O(N+mn) \leq O(N+N^2) \approx O(N^2)$ to complete.

2.6 Experiments and Results

To assess the new encoding in this paper, three experiments with results are presented: a comparison to prior SMT encodings on a zero-buffer semantics, a scalability study on the effects of non-determinism in the execution time on infinite buffer semantics, and an evaluation on typical benchmark programs again with infinite buffer semantics. All of the experiments use the Z3 SMT solver ([13]) and are measured on a 2.40 GHz Intel Quad Core processor with 8 GB memory running Windows 7.

The initial program trace for the experiments is generated using the MCA provided reference solution with fixed input. In other words, the only non-determinism in the programs is that allowed by the MCAPI specification. As such, the experiments only consider

one path of control flow through the program. Complete coverage of the program for verification purposes would need to generate input to exercise different control flow paths. Where appropriate, the time to generate the match pair sets from the input trace is reported separately.

2.6.1 Comparison to Prior SMT Encoding

To our best knowledge, the current most effective SMT encoding for verification of message passing program traces is the order-based encoding that describes the happens-before relation directly in the encoding and is only functional for zero-buffer semantics in its current form [17]. The order-based encoding is more complex than the encoding in this paper and generates more clauses for the SMT solver. Although the tool to generate the encoding is not publicly available, the authors of the order-based encoding graciously encoded several contrived benchmarks used for correctness testing. These benchmarks are best understood as *toy* examples that plumb the MCAPI semantics to clarify intuition on expected behavior.

The zero-buffer encoding in this paper is compared directly to the order-based encoding on the contrived benchmarks. The order-based encoding yields incorrect answers for several programs. Where the order-based encoding returns correct answers, the new encoding, on average, requires 70% fewer clauses, uses half the memory as reported by the SMT solver, and runs eight times faster. The dramatic improvement of the new encoding over the order-based encoding is a direct result of the match pairs that simplify the happens-before constraints and avoids redundant constraints in the transitive closure of the happens-before relation.

2.6.2 Scalability Study

The intent of the scalability study is to understand how performance is affected by the number of messages in the program trace and the level of non-determinism in choosing match pairs where multiple sends are able to match to multiple receives. The programs for

Table 2.1: Scaling as a function of non-determinism

Test Programs		Performance	
N	Feasible Sets	Time (hh:mm:ss)	Memory(MB)
30	30!(∼3E32)	00:00:36	20.11
40	40!(∼8E47)	00:03:22	47.12
50	50!(∼3E64)	00:16:11	102.65
60	60!(∼8E81)	00:47:29	189.53
70	70!(∼1E100)	02:00:30	364.25

this study consist of a simple pattern of a single thread to receive messages and N threads to send messages. The single thread sequentially receives N messages containing integer values and then asserts that every message did not receive a specific value. In other words, a violation is one where each message has a specific value. The remaining N threads send a message, each containing a different unique integer value, to the single thread that receives. These programs represent the worst-case scenario for non-determinism in a message passing program as any send is able to match with any receive in the runtime, and the assertion is only violated when each send is paired with a specific receive. The SMT solver must search through the multitude of match pairs, $N \times N$, to find the single precise subset of match pairs that triggers the violation. In this program structure, there are $N!$ feasible ways to match N sends to N receives.

The study takes an initial program of $N = 30$, so 31 threads, and varies N to see how the SMT solver scales. A small N is an easy program while a large N is a hard program. Table 2.1 shows how the new encoding scales with hardness. The first column is the number of messages, or N, and the second column is the number of feasible match pair subsets that correctly match every receive to a unique send. As expected, running time and memory consumption increase non-linearly with hardness.

The case where $N = 70$ represents having 70 concurrent messages in flight from 70 different threads of execution. Such a scenario is not entirely uncommon in a high performance computing application, and it appears the new encoding is able to reasonably

scale to such a level of concurrency. The result provides a bound on expected cost for analysis given the message passing behavior in a program. It is expected that the analysis of any program with fewer than 70! possible choices of feasible match pair resolutions will complete in a reasonable amount of time. Regardless, such a high-level concurrency seems unlikely in the embedded space to which MCAPI is targeted.

2.6.3 Typical Benchmark Programs

The results in the prior section suggest that the number of messages is not the deciding factor in hardness for the new encoding; rather, hardness is measured by the number of feasible match pair sets. This section further explores the observation to show that some programs are easy, even if there are many messages, while other programs are hard, even though there are only a few messages.

The goal of these experiments is to measure the new encoding on several benchmark programs. MCAPI is a new interface, and to date, the authors are not aware of publicly available programs written against the interface aside from the few toy programs that come with the library distribution. As such, the benchmarks in the experiments come from a variety of sources.

- *LE* is the leader election problem and is common to benchmarking verification algorithms.
- *Router* is an algorithm to update routing tables. Each router node is in a ring and communicates only with immediate neighbors to update the tables. The program ends when all the routing tables are updated.
- *MultiM* is an extension to a program in the MCAPI library distribution and is similar to the program in Figure 2.9. The extension adds extra iterations to the original program execution to generate longer execution trace.
- *Pktuse* is a benchmark from the MPI test suite [40]. The program creates 5 tasks—each of which randomly sends several messages to the other tasks.

Table 2.2: Performance on selected benchmarks

Test Programs			Performance			
Name	# Mesg	Feasible Sets	EG(s)	MG(s)	Time (hh:mm:ss)	Memory(MB)
LE	620	1	1.49	0.051	<00:00:01	33.41
Router	200	∼6E2	0.417	0.032	00:00:02	15.03
MultiM	100	∼1E40	0.632	0.436	00:16:40	135.19
Pktuse	512	∼1E81	10.190	9.088	02:06:09	1539.90

The benchmark programs are intended to cover a spectrum of program properties. As before, the primary measure of hardness in the programs in not the number of messages but rather the size of the match pair set and the number of feasible subsets. The *LE* program is the easiest program in the suite. Although it sends 620 messages, there is only a single feasible match pair set. The programs *Router*, *MultiM*, and *Pktuse* respectively increase in hardness, which again is not related to the total number of messages but rather the total number of feasible match-sets that must be considered. For example, even though *Router* has 200 messages, it is an easier problem that *MultiM* that has 100 messages. The *Pktuse* program does have the most number of messages, 512, and in this case, the largest number of feasible match pair sets.

Table 2.2 shows the results for the benchmark suite. Other than the metrics used in Table 2.1, the time of generating the encoding and the match pairs is included in the third and fourth columns respectively. Note that the time shown in the third column includes the time in the fourth column. As before, the running time tracks hardness and not the total number of messages. The table also shows the cost of match pair generation as it dominates the encoding time for the *Pktuse* program (an item for future work).

The benchmark suite demonstrates that a message passing program may have a large degree of non-determinism in the runtime that is prohibitive to verification approaches that directly enumerate non-determinism such as a model checker. The SMT encoding, however, pushes the problem to the SMT solver by generating the possible match pairs and then relying on advances in SMT technology to resolve the non-determinism in a way that vio-

lates the assertion. Of course, the SMT problem itself is NP-complete, so performance is only reasonable for small problem instances. The benchmark suite suggests that problem instances with astonishingly large numbers of feasible match pair sets are able to complete in a reasonable amount of time using the new encoding in this paper; though, the time to generate the match pairs may quickly become prohibitive.

2.7 Related Work

Morse *et al.* provided a formal modeling paradigm that is callable from the C language for the MCAPI interface [38]. This model correctly captures the behavior of the interface and can be applied to model checking C programs that use the API. The work is a direct application of model checking and directly enumerates the non-determinism in the runtime to construct an exhaustive proof. The SMT encoding in this paper pushes that complexity to the SMT solver and leverages recent advances in SMT technology to find a satisfying assignment.

Sharma *et al.* present an dynamic model checker for MCAPI programs built on top of the MCA provided MCAPI runtime [48]. MCC systematically enumerates all non-determinism in the MCAPI runtime under zero-buffer semantics. It employs a novel dynamic partial order reduction to avoid enumerating redundant message orders. This work claims SMT technology is more efficient in practice in resolving non-determinism in a away to violate correctness properties.

Wang *et al.* present an SMT encoding for shared memory semantics for a given input trace from a multi-threaded program [64]. As mentioned previously, the program is partitioned into several concurrent trace programs, and the encoding for each program is verified using SMT technology. Elwakil *et al.* extend the encoding to message passing programs using the MCAPI semantics [17, 18]. The comparison to the encoding in this work is already discussed previously.

An important body of work is being pursued for MPI program verification [19, 51, 53–55, 59, 61]. Highlights include an extension to the SPIN model checker for MPI programs, symbolic execution tools for MPI programs including new approaches to computing loop invariants, and various dynamic verification tools for MPI programs. Although MPI is more expressive than MCAPI, the correctness properties in MCAPI are similar to those in MPI. More importantly, the encoding in this work should be applicable to MPI programs that do not include collective operations. An important aspect of future work is to extend the encoding to collectives.

There is a rich body of literature for SMT/SAT based Bounded Model Checking. Burckhardt *et al.* exhaustively check all executions of a test program by translating the program implementation into SAT formulas [7]. The approach relies on counter-examples from the solvers the refine the encoding. The SMT encoding in this work is able to directly resolve the match-pair set over-approximation directly without needing to check a counter-example.

Dubrovin *et al.* give a method to translate an asynchronous system into a transition formula over three partial order semantics [15]. The encoding adds constraints to compress the search space and decrease the bound on the program unwinding. The encoding in this paper operates on a program execution and does not need to resolve a bound.

Kahlon *et al.* presented a partial order reduction, *MPOR*, that operates in the bounded model checking space [31]. It guarantees that exactly one execution is calculated per each Mazurkiewicz trace to reduce the search space. It would be interesting to see if MPOR is able to extend to message passing semantics. Other work in bounded model checking explores heap-manipulating programs and challenges in sequential systems code [33, 34].

The application of static analysis is another interesting thread of research to test or debug message passing programs with some work in the MPI domain [6, 21, 66]. The work

is important as it lays the foundation for refining match-pair sets to only include those that cannot be statically pruned.

2.8 Conclusions and Future Work

This paper presents a proof that the problem of resolving non-determinism in message passing in a way that meets asserts is NP-complete. The paper then presents an SMT encoding of an MCAPI program execution that uses match pairs directly rather than the state-based or order-based encoding in the prior work. The encoding is generated from a given execution trace and a set of potential match pairs that can be over-approximated. The encoding takes extra care in forming the SMT problem to preclude bogus match pairs in any over-approximation of the match pair input set. Critically, the encoding is the first to correctly capture the non-deterministic behaviors of an MCAPI program execution under infinite-buffer semantics.

This paper further defines an algorithm with $O(N^2)$ time complexity to over-approximate the true set of match pairs, where N is the total number of code lines of the program. A comparison to prior work, [17], for a set of "toy" examples under zero-buffer semanics shows the new encoding capable and efficient in capturing correct behaviors of an MCAPI program execution. Experiments further show that the encoding scales to programs with significant levels of non-determinism in how sends match to receives.

The results show that a large match-pair set does affect the runtime performance of the encoding in the SMT problem even if the encoding is sound under an over-approximation. Future work explores new methods for generating a much more precise set of match pairs. The encoding is dependent on an input execution trace of the program. Future work explores integrating the encoding into a model checker. The model checker generates a program trace that is encoded and verified. The result is then used to inform the model checker as to where it needs to backtrack to generate a new execution trace. The goal is to use the trace verification to construct a better partial order reduction in the model checker.

Finally, given the importance of high performance computing, future work looks to extend the encoding to account for MPI collective operations. This direction is motivated by the results where the encoding seems to scale to significant levels of concurrency. It should be possible to express MPI collectives as additional constraints in the encoding and apply the technique to MPI programs directly.

Chapter 3

Detecting MPI Zero Buffer Incompatibility by SMT Encoding

A prevalent asynchronous message passing standard is the Message Passing Interface (MPI). There are two runtime semantics for MPI: zero buffer (messages have no buffering) and infinite buffer (messages are copied into a runtime buffer on the API call). A problem in any MPI program, intended or otherwise, is zero buffer incompatibility. A zero buffer incompatible MPI program deadlocks. This problem is difficult to predict because a developer does not know if the deadlock is based on the buffering semantics or a bad program. This paper presents an algorithm that encodes a single-path MPI program as a Satisfiability Modulo Theories (SMT) problem, which if satisfiable, yields a feasible schedule, such that it proves the program is zero buffer compatible. This encoding is also adaptable to checking assertion violation for correct computation. To support MPI semantics, this algorithm correctly defines the point-to-point communication and collective communication with respective rules for both infinite buffer semantics and zero buffer semantics. The novelty in this paper is considering only the schedules that strictly alternate sends and receives leading to an intuitive zero buffer encoding. This paper proves that the set of all the strictly alternating schedules is capable of covering all the message communication that may occur in any execution under zero buffer semantics. Experiments demonstrate that the SMT encoding is correct and highly efficient for a set of benchmarks compared with two state-of-art MPI verifiers.

3.1 Introduction

Message passing technology has become widely used in many fields such as medical devices and automobile systems. The Message Passing Interface (MPI) plays a significant role as a common standard. It is easy for a developer to implement a message passing scenario using MPI semantics, including:

- zero buffer semantics (messages have no buffering) and infinite buffer semantics (messages are copied into a runtime buffer on the API call) [59],
- MPI point-to-point operations (e.g., send and receive), and
- MPI collective operations (e.g., barrier and broadcast).

A problem in any MPI program is zero buffer incompatibility. A zero buffer incompatible program deadlocks. If there exists any feasible schedule for a program under zero buffer semantics, this program is zero buffer compatible. Note that the zero buffer incompatibility is not equivalent to deadlock that may be caused by reasons other than buffering. The zero buffer incompatibility is essential to any MPI application since it is difficult for a developer to predict. This problem is also very difficult to verify because of the complicated MPI semantics. In particular, the message passing may be non-deterministic such that a receive may match more than one send in the runtime system. Also, the MPI collective operations that synchronize the program may change how messages communicate. To address the problem of zero buffer incompatibility, this paper presents an algorithm that encodes a single-path MPI program into a Satisfiability Modulo Theories (SMT) problem [4]. This encoding is resolved by a standard SMT solver in such a way that the program is proved/disproved to be zero buffer compatible. This encoding is also adaptable to checking assertion violation caused by message non-determinism.

Several solutions were proposed to verify MPI programs. The POE algorithm is capable of dynamically analyzing the behavior of an MPI program [58]. This algorithm is implemented by a modern MPI verifier, ISP. As far as we know, there is no research proposed merely for zero buffer incompatibility. Though the works on MPI deadlock are also capable

of detecting zero buffer incompatibility, they suffer from the scalability problem [20, 49]. In particular, the algorithm MSPOE is an extension of POE. It is able to detect deadlock in an MPI program [49]. Forejt et al. proposed a SAT based approach to detect deadlock in a single-path MPI program [20].

The problem of generating a input match pair set is NP-Complete. The preprocessing, however, only needs to over-approximate the match pairs in quadratic time complexity. Before discussing how the new algorithm is capable of detecting zero buffer incompatibility, this paper needs to present in detail the complete list of encoding rules for several MPI operations, including a few rules that are trivial to define. In particular, the MPI non-deterministic point-to-point communication is similar to how message communicate in the Multicore Communications API (MCAPI). As such, this paper adapts the existing encoding rules for MCAPI defined in prior work [28]. This paper also presents how to encode deterministic receive operations and collective operations, which are essential to MPI semantics, into a set of SMT formulas. The formula size is quadratic. Note that the prior work also provides a list of non-intuitive and complicated zero buffer encoding rules. However, these rules are only useful for manually encoding the zero buffer semantics. The new zero buffer encoding in this paper considers only the schedules that strictly alternate sends and receives, therefore, not only is it correct and intuitive, it is easy to build automatically. The use of strict alternation is able to cover any message communication that may occur in any execution under zero buffer semantics. This strategy is inspired by Threaded-C Bounded Model Checking (TCBMC) that extends C Bounded Model Checking (CBMC) [10, 11] to support concurrent C program verification [44]. It assumes each lock operation and its paired unlock operation is ordered alternatingly in any execution.

To summarize, the main contributions of this paper include,

- a new zero buffer encoding with strict alternation of sends and receives that is capable of detecting zero buffer incompatibility,

- a new encoding algorithm that supports MPI deterministic point-to-point communication and MPI collective communication, and
- a set of benchmarks that demonstrate the new encoding is more efficient than two state-of-art MPI verifiers.

The rest of the paper is organized as follows: Sections 2 and 3 present a few trivial rules for encoding MPI operations, including the summarization of the prior work [28] in section 2 and the rules for MPI deterministic operations and collective operations in section 3; Based on these rules, section 4 discusses the new zero buffer encoding and how it is able to check zero buffer incompatibility; Section 5 gives the experiment results; Section 6 discusses the related work; and Section 7 discusses the conclusion and future work.

3.2 SMT Encoding for MCAPI

This section summarizes the SMT encoding rules (except the rules for zero buffer semantics) discussed in the prior work for MCAPI verification. MCAPI is a light-weight message passing interface that only uses sends and wildcard receives for message communication. A wildcard receive is a receive that can match sends from any source. These rules are used to encode MPI non-deterministic point-to-point communication. In general, the SMT encoding is generated from 1) an execution trace of a program that includes a sequence of events; and 2) a set of possible match pairs for message communication. Intuitively, a match pair is a coupling of a send and a receive.

The encoding contains a timestamp $time_{\mathtt{e}}$ for every event `e` in a program. Intuitively, the timestamp is an integer. The event order is enforced by the *Happens-Before* (`HB`) operator, denoted as $\prec_{\mathtt{HB}}$, over two events, `e1` and `e2` respectively, such that $time_{\mathtt{e1}} < time_{\mathtt{e2}}$ holds. The send and receive operations are encoded as tuples. In particular, a send operation $\mathtt{S} = (M_{\mathtt{S}}, time_{\mathtt{S}}, e_{\mathtt{S}}, value_{\mathtt{S}})$, is a four-tuple of variables. $M_{\mathtt{S}}$ is the timestamp of the matching receive; $time_{\mathtt{S}}$ is the timestamp of `S`; $e_{\mathtt{S}}$ is the destination endpoint of a message;

and $value_{\mathtt{S}}$ is the transmitted value. The values of $e_{\mathtt{S}}$ and $value_{\mathtt{S}}$ are fixed from the input trace. Similarly, a receive operation $\mathtt{R} = (M_{\mathtt{R}}, time_{\mathtt{R}}, e_{\mathtt{R}}, value_{\mathtt{R}}, time_{\mathtt{W_R}})$, is modeled by a five-tuple of variables. $M_{\mathtt{R}}$ is the timestamp of the matching send; $time_{\mathtt{R}}$ is the timestamp of $\mathtt{R}$; $e_{\mathtt{R}}$ is the destination endpoint of a message; $value_{\mathtt{R}}$ is the received value; and $time_{\mathtt{W_R}}$ is the timestamp of the nearest-enclosing wait $\mathtt{W_R}$. A nearest-enclosing wait is a wait that witnesses the completion of a receive by indicating that the message is delivered and that all the previous receives on the same process issued earlier are completed as well. The value of $e_{\mathtt{R}}$ is fixed from the input trace. Note that only wildcard receive is used in the MCAPI semantics. Therefore, the encoding for MCAPI does not need to specify the message source endpoints in sends and receives. The message communication topology is encoded as a set of match pairs defined in Definition 15.

Definition 15 (Match Pair)**.** *A match pair,* $\langle \mathtt{R}, \mathtt{S} \rangle$*, for a receive* $\mathtt{R} = (M_{\mathtt{R}}, time_{\mathtt{R}}, e_{\mathtt{R}}, value_{\mathtt{R}}, time_{\mathtt{W_R}})$ *and a send* $\mathtt{S} = (M_{\mathtt{S}}, time_{\mathtt{S}}, e_{\mathtt{S}}, value_{\mathtt{S}})$*, corresponds to the following constraints:*

1. $M_{\mathtt{R}} = time_{\mathtt{S}} \wedge M_{\mathtt{S}} = time_{\mathtt{R}}$
2. $e_{\mathtt{R}} = e_{\mathtt{S}} \wedge value_{\mathtt{R}} = value_{\mathtt{S}}$
3. $time_{\mathtt{S}} \prec_{\mathtt{HB}} time_{\mathtt{W_R}}$

We define the potential sends for a receive $\mathtt{R}$, denoted as Match($\mathtt{R}$), as the set of all the sends that $\mathtt{R}$ may potentially match. The encoding rules are given in Figure 3.1: rules (3.1) – (3.5) encode the program order; rule (3.6) encodes the match pairs; and rules (3.7) and (3.8) encode the assumptions on control flow and the negated assertion respectively. Assume a program contains $\mathcal{N}$ API calls, the generated SMT encoding contains $\mathcal{O}(\mathcal{N}^2)$ formulas. The following sections discuss the extension to MPI semantics and more importantly zero buffer incompatibility that is not included in the prior work.

– Sequential sends $\mathtt{S}$ and $\mathtt{S}'$ from a common source to a common destination $e_{\mathtt{S}} = e_{\mathtt{S}'}$	$time_{\mathtt{S}} \prec_{\mathtt{HB}} time_{\mathtt{S}'}$	(3.1)
– Sequential receives $\mathtt{R}$ and $\mathtt{R}'$, on a common process, $e_{\mathtt{R}} = e_{\mathtt{R}'}$	$time_{\mathtt{R}} \prec_{\mathtt{HB}} time_{\mathtt{R}'}$	(3.2)
–Receive $\mathtt{R}$ and its nearest-enclosing wait $\mathtt{W_R}$	$time_{\mathtt{R}} \prec_{\mathtt{HB}} time_{\mathtt{W_R}}$	(3.3)
– Sequential order over a nearest-enclosing wait $\mathtt{W_R}$ for a receive $\mathtt{R}$ and a send $\mathtt{S}$	$time_{\mathtt{W_R}} \prec_{\mathtt{HB}} time_{\mathtt{S}}$	(3.4)
– Two Sends, $\mathtt{S}$ and $\mathtt{S}'$, to a common destination, $e_{\mathtt{S}} = e_{\mathtt{S}'}$, such that $time_{\mathtt{S}} \prec_{\mathtt{HB}} time_{\mathtt{S}'}$ is enforced	$M_{\mathtt{S}} \prec_{\mathtt{HB}} M_{\mathtt{S}'}$	(3.5)
–Match pairs for any receive $\mathtt{R}$	$\bigvee_{\mathtt{S} \in \mathrm{Match}(\mathtt{R})} \langle \mathtt{R}, \mathtt{S} \rangle$	(3.6)
–Assumption $\mathtt{A}$ on control flow	$\mathtt{A}$	(3.7)
–User provided assertion $\mathtt{P}$	$\neg \mathtt{P}$	(3.8)

Figure 3.1: SMT encoding for MPI non-deterministic point-to-point communication.

3.3 Extension to MPI

This section discusses the new encoding for MPI deterministic point-to-point communication and collective communication. To be precise, the encoding needs to add variables $src_{\mathtt{S}}$ and $src_{\mathtt{R}}$ to the send operation and the receive operation respectively. Intuitively, the variables $src_{\mathtt{S}}$ and $src_{\mathtt{R}}$ are the source endpoints of messages. As such, a send operation $\mathtt{S} = (M_{\mathtt{S}}, time_{\mathtt{S}}, e_{\mathtt{S}}, src_{\mathtt{S}}, value_{\mathtt{S}})$, is now a five-tuple of variables. A receive operation $\mathtt{R} = (M_{\mathtt{R}}, time_{\mathtt{R}}, e_{\mathtt{R}}, src_{\mathtt{R}}, value_{\mathtt{R}}, time_{\mathtt{W_R}})$, is now a six-tuple of variables. We constrain the variable $src_{\mathtt{R}}$ to be equal to $*$ for a wildcard receive $\mathtt{R}$. In addition, the match pair defined in Definition 15 adds a new constraint:

$$src_{\mathtt{R}} = * \vee src_{\mathtt{S}} = src_{\mathtt{R}},$$

indicating that either $\mathtt{R}$ is a wildcard receive or the source endpoints are matched for $\mathtt{S}$ and $\mathtt{R}$.

As discussed earlier, collective operations are used to synchronize an MPI program. To be precise, collective operations such as barriers block the execution of processes until all the

members in a group are matched. In addition, some type of collective operations such as broadcast are able to send internal messages amongst its group, and/or to execute global operations. MPI semantics guarantee that messages generated on behalf of collective operations are not confused with messages generated by point-to-point operations. Therefore, the encoding in this paper puts emphasis on how to reason about the synchronization of collective operations as it affects point-to-point communication. The internal message passing and the execution of global operations by collective communication can be added as SMT constraints to the encoding. In the following discussion, we take barrier as an example. The barrier is defined as a group in Definition 16.

Definition 16 (Barrier)**.** *The occurrence of a group of barriers,* $\mathtt{B} = \{\mathtt{B}_0, \mathtt{B}_1, ..., \mathtt{B}_n\}$*, is captured by a single timestamp,* $time_{\mathtt{B}}$*, that marks when all the members in the group are matched.*

Even though barriers affect the issuing order of two events, it is hard to determine whether they prevent a send from matching a receive. As an example, the message "1" in Figure 3.2 may flow into R even though R is ordered before the barrier and S is ordered after the barrier. The wait W(&h2) determines the behavior. If the program had issued W(&h2) before the barrier, R would have to be completed before the barrier, meaning the message is delivered. The encoding further defines the nearest-enclosing barrier (Definition 17) for this type of interaction.

Definition 17 (Nearest-Enclosing Barrier)**.** *For any process* i*, a receive* R *has a nearest-enclosing barrier* B *if and only if*

1. *the nearest-enclosing wait,* W*, of* R *is ordered before* $\mathtt{B}_i \in \mathtt{B}$*, and*

Process 0	Process 1
B(comm)	R(from P0, A, &h2)
S(to P1, "1", &h1)	B(comm)
W(&h1)	W(&h2)

Figure 3.2: Message Communication with Barriers.

– Any receive R that has a nearest-enclosing barrier B and a nearest-enclosing wait $\mathtt{W_R}$ — $time_{\mathtt{W_R}} \prec_{\mathtt{HB}} time_{\mathtt{B}}$ (3.9)

– Any barrier B and any operation O ordered after a member of B — $time_{\mathtt{B}} \prec_{\mathtt{HB}} time_{\mathtt{O}}$ (3.10)

Figure 3.3: SMT encoding for MPI collective communication.

2. *there does not exist any receive* $\mathtt{R}'$ *on process* i*, with a nearest-enclosing wait,* $\mathtt{W}'$*, such that 1)* $\mathtt{W}'$ *is ordered after* W*; and 2)* $\mathtt{W}'$ *is ordered before* $\mathtt{B_i}$*.*

Based on the definitions above, the encoding defines two rules for program order in Figure 3.3. Rule (3.9) only constrains the program order over the nearest-enclosing wait and the nearest-enclosing barrier for a receive. The order over this receive and the nearest-enclosing barrier is not constrained. For rule (3.10), a barrier has to happen before any operation ordered after it.

3.4 Zero Buffer Incompability

This section presents a new zero buffer encoding that is easy to build automatically. The key insight is to order a send immediately preceding the matching receive in a match pair captured in Definition 18.

Definition 18 (Match Pair *)**.** *A match pair,* $\langle \mathtt{R}, \mathtt{S} \rangle^*$*, for a receive* $\mathtt{R} = (M_{\mathtt{R}}, time_{\mathtt{R}}, e_{\mathtt{R}}, src_{\mathtt{R}}, value_{\mathtt{R}}, time_{\mathtt{W_R}})$ *and a send* $\mathtt{S} = (M_{\mathtt{S}}, time_{\mathtt{S}}, e_{\mathtt{S}}, src_{\mathtt{S}}, value_{\mathtt{S}})$*, corresponds to the constraints:*

1. $M_{\mathtt{R}} = time_{\mathtt{S}} \wedge M_{\mathtt{S}} = time_{\mathtt{R}}$
2. $e_{\mathtt{R}} = e_{\mathtt{S}} \wedge value_{\mathtt{R}} = value_{\mathtt{S}}$
3. $src_{\mathtt{R}} = * \vee src_{\mathtt{S}} = src_{\mathtt{R}}$
4. $\boldsymbol{time_S = time_R - 1}$

Intuitively, the consecutive order over a send and the matching receive is defined in the bold rule 4 of Definition 18. Any resolved execution strictly alternates sends and receives.

– Sequential sends $\mathtt{S}$ and $\mathtt{S}'$ from a common source to any destinations	$time_{\mathtt{S}} \prec_{\mathtt{HB}} time_{\mathtt{S}'}$	(3.1*)
– Match pairs for any receive $\mathtt{R}$	$\bigvee_{\mathtt{S}_i \in \mathrm{Match}(\mathtt{R})} \langle \mathtt{R}, \mathtt{S}_i \rangle^*$	(3.6*)
– Send $\mathtt{S}$ and receive $\mathtt{R}$ are in a sequential order on a common process	$time_{\mathtt{S}} \prec_{\mathtt{HB}} time_{\mathtt{R}}$	(3.11)
– Send $\mathtt{S}$ and barrier $\mathtt{B}$ are in a sequential order on a common process	$time_{\mathtt{S}} \prec_{\mathtt{HB}} time_{\mathtt{B}}$	(3.12)

Figure 3.4: SMT encoding for zero buffer semantics.

Definition 19 (Strict Alternation)**.** *A set of sends,* $\mathbb{S}$*, and a set of receives,* $\mathbb{R}$*, are strictly alternated if and only if each send in* $\mathbb{S}$ *immediately precedes the matching receive in* $\mathbb{R}$ *and each receive in* $\mathbb{R}$ *immediately follows the matching send in* $\mathbb{S}$*.*

To further constrain the program order for zero buffer semantics, new rules are added as shown in Figure 3.4: on each process a send happens before a receive that is issued later (rule (3.11)); and similarly, on each process a send happens before a barrier that is issued later (rule (3.12)). In addition, Rule (3.1*) replaces rule (3.1) as zero buffer semantics do not allow a new send to be issued before the pending send is completed on a common process. Rule (3.6*) replaces rule (3.6) to enforce strict alternation for every send and its matching receive.

To check zero buffer incompatibility, the encoding intends to find a feasible strictly alternating schedule by constraining the over-approximated match relation and the program order. If no feasible schedule exists, meaning there is no ordering that satisfies the happens-before relation, the program is zero buffer incompatible, and it deadlocks under zero buffer semantics; otherwise, zero buffer compatibility is proved. Since this process only relies on the event ordering, the constraints of user-provided assertions defined in rule (3.8) are removed.

Notice that a program with deadlock may be zero buffer compatible. Intuitively, a deadlock can be caused by an orphaned send or receive that is never matched, a depen-

dency circuit existing in the message communication topology, the improper use of collective operations, etc.

3.4.1 Correctness

As discussed earlier, the zero buffer encoding only considers schedules that strictly alternate sends and receives, therefore, it makes the encoding rules intuitive and easy to build automatically. The fundamental insight is that this strict alternation is sufficient to cover all possible resolutions of message communication. We prove a theorem later that asserts this insight. Before that, we need to define a few terms.

Definition 20 (Method Invocation)**.** *A invocation of a method,* `M`*, with a list of specific values of arguments,* $(args\ \cdots)$*, on process* P*, denoted as* $P : \mathtt{M}_i(args\ \cdots)$*, is a event that occurs when* `M` *is invoked.*

Definition 21 (Method Response)**.** *A response of a method,* `M`*, with a specific return value,* $resp$*, on process* P*, denoted as* $P : \mathtt{M}_r(resp)$*, is a event that occurs when* `M` *returns.*

Based on Definition 20 and Definition 21, an operation is split into two events: invocation and response. The invocation asserts the issuing of an operation with concrete arguments. We use the notations $\mathbb{S}_i$ and $\mathbb{R}_i$ to represent the set of all the send invocations and the set of all the receive invocations respectively for an MPI program. The response asserts the completion of an operation with a concrete return value. We use the notations $\mathbb{S}_r$ and $\mathbb{R}_r$ to represent the set of all the send responses and the set of all the receive responses respectively for an MPI program. A history of an MPI program relies on method invocation and method response.

Definition 22 (History)**.** *For an MPI program, let* $\mathcal{H}$ *be a history with a total order over method invocations and method responses for a set of send operations,* $\mathbb{S}$*, a set of receive operations,* $\mathbb{R}$*, and a set of barriers,* $\mathbb{B}$*.*

Based on Definition 22, we further define a legal history as follows.

Definition 23 (Legal History)**.** *A history, $\mathcal{H}$, for an MPI program is legal if the total order over all the events in $\mathcal{H}$ is allowed by MPI semantics.*

A legal history defined in Definition 23 represents a total order over events for an MPI program. A legal history only takes care of sends, receives and barriers because only they matter to message communication. In other words, the events in a legal history can be used to evaluate how messages may flow in a runtime system. Since the arguments are concrete in any method invocation and the return value is also concrete in any method response, a legal history corresponds to a precise resolution of message communication. To find a feasible message communication for a receive R, one only needs to search through the preceding events and find a send S that matches the endpoints of R and obeys the non-overtaking order for all the sends to the common destination endpoint. The legal history asserts that the total order over all the events is allowed by MPI semantics. In particular, if a legal history is allowed by zero buffer semantics, we call it a zero-buffer legal history.

To prove the theorem later, we need to compare two legal histories for equivalence. The equivalence relation relies on the following definitions for projections.

Definition 24 (Projection To Process)**.** *A projection of a legal history, $\mathcal{H}$, to a process, P, denoted as $\mathcal{H}|_P$, is a sequence of all the events on process P in $\mathcal{H}$, such that the order over any pair of events in $\mathcal{H}|_P$ is identical as in $\mathcal{H}$.*

Definition 25 (Projection To Receive Response)**.** *A projection of a legal history, $\mathcal{H}$, to the receive responses, $\mathbb{R}_r$, denoted as $\mathcal{H}|_{\mathbb{R}_r}$, is a sequence of receive responses in $\mathcal{H}$ such that the order over any pair of receive responses in $\mathcal{H}|_{\mathbb{R}_r}$ is identical as in $\mathcal{H}$.*

Based on Definition 24 and Definition 25, a legal history can be further projected to the receive responses $\mathbb{R}_r$ on process P. We use $\mathcal{H}|_{\mathbb{R}_r,P}$ to represent this projection. The equivalence relation relies on $\mathcal{H}|_{\mathbb{R}_r,P}$.

Definition 26 (Equivalence Relation)**.** *Two legal histories for an MPI program, say $\mathcal{H}$ and $\mathcal{L}$ respectively, are equivalent, denoted as $\mathcal{H} \sim \mathcal{L}$, if and only if their projections to the receive*

responses $\mathbb{R}_r$ on each single process P, $\mathcal{H}|_{\mathbb{R}_r,P}$ and $\mathcal{L}|_{\mathbb{R}_r,P}$ respectively, agree on the return values of $\mathbb{R}_r$.

The following lemma is essential to proving that Definition 26 is a valid equivalence relation. Definition 27 defines the reachable set of legal histories for an MPI program.

Definition 27 (Reachable Legal Histories)**.** *For an MPI program, p, let $\mathcal{RS}(p)$ be a set of all the legal histories that are reachable from p.*

Lemma 3 (Validity of Equivalence Relation)**.** *The $\sim$-operator is an equivalence relation over the set of all legal histories.*

Proof. Proof by the definition of equivalence relation. Consider three legal histories, $\mathcal{H}, \mathcal{L}, \mathcal{T} \in \mathcal{RS}(p)$, for an MPI program, p, the equivalence relation in Definition 26 is reflexive, symmetric and transitive.

1. $\mathcal{H} \sim \mathcal{H}$. The reflexivity is true because the projection $\mathcal{H}|_{\mathbb{R}_r,P}$ to the receive responses $\mathbb{R}_r$ on any process P, and the projection itself agree on the return values of $\mathbb{R}_r$.
2. $\mathcal{H} \sim \mathcal{L}$ then $\mathcal{L} \sim \mathcal{H}$. The symmetry is also true because $\mathcal{H} \sim \mathcal{L}$ and $\mathcal{L} \sim \mathcal{H}$ both indicate that the projections $\mathcal{H}|_{\mathbb{R}_r,P}$ and $\mathcal{L}|_{\mathbb{R}_r,P}$ to the receive responses $\mathbb{R}_r$ on any process P agree on the return values of $\mathbb{R}_r$.
3. $\mathcal{H} \sim \mathcal{L}$ and $\mathcal{L} \sim \mathcal{T}$ then $\mathcal{H} \sim \mathcal{T}$. As for the transitivity, for all the receive responses $\mathbb{R}_r$ on any process P in the MPI program, $\mathcal{H} \sim \mathcal{L}$ indicates that the projections $\mathcal{H}|_{\mathbb{R}_r,P}$ and $\mathcal{L}|_{\mathbb{R}_r,P}$ to the receive responses $\mathbb{R}_r$ on any process P agree on the return values of $\mathbb{R}_r$. Further, $\mathcal{L} \sim \mathcal{T}$ indicates that $\mathcal{L}|_{\mathbb{R}_r,P}$ is also identical to $\mathcal{T}|_{\mathbb{R}_r,P}$. Therefore, $\mathcal{H}|_{\mathbb{R}_r,P}$ is identical to $\mathcal{T}|_{\mathbb{R}_r,P}$. Since P is an arbitrary process in the MPI program and $\mathbb{R}_r$ do not change on P, thus $\mathcal{H} \sim \mathcal{T}$ is implied.

Based on the reflexivity, symmetry and transitivity, this equivalence relation is able to partition the reachable set of legal histories, $\mathcal{RS}(p)$, and therefore identifies the equivalent classes among $\mathcal{RS}(p)$. □

Based on Lemma 3, we further use $\mathcal{E}(\mathcal{RS}(p))$ to represent the equivalent classes for all the legal histories in $\mathcal{RS}(p)$. If an equivalent class includes zero-buffer legal histories, we call it a zero-buffer available equivalent class. The following theorem states that a representative exists for each zero-buffer available equivalent class.

Theorem 5. *For any MPI program, p, each zero-buffer available equivalent class, $E \in \mathcal{E}(\mathcal{RS}(p))$, has a representative zero-buffer legal history, T, that strictly alternates the sends, $\mathbb{S}$, and the receives, $\mathbb{R}$.*

Proof. Proof by showing the existence of a zero-buffer legal history for any equivalence class. First, assume there is a legal history $\mathcal{L} \in E$. Second, assume $\mathbb{R}_r$ is a set of all the receive responses. The projection $\mathcal{L}|_{\mathbb{R}_r}$ is a sequence of receive responses that reflects how messages are received in $\mathcal{L}$ for all the processes. Since the message communication is precisely resolved in $\mathcal{L}$, each receive in $\mathbb{R}$ is matched with a send in $\mathbb{S}$. Based on $\mathcal{L}$ and $\mathcal{L}|_{\mathbb{R}_r}$, a new sequence, $\mathcal{T}$, can be produced by two steps: 1) inserting the corresponding receive invocation immediately preceding each receive response; and 2) inserting the invocation and the response of the matching send immediately preceding each receive invocation. Based on those steps, $\mathcal{T}$ strictly alternates $\mathbb{S}$ and $\mathbb{R}$. Further, it obeys the conditions in Definition 23: first, the consecutive order over a send and the matching receive in $\mathcal{L}$ still exists in $\mathcal{T}$; second, if the matching receive is issued earlier on process P, there is no way to execute any operation after the receive on process P until it is matched with a send, therefore, postponing issuing the receive after the matching send does not violate the MPI semantics. Under zero buffer semantics, it is not possible to order a send and the matching receive other than the two stated situations above. Notice that $\mathcal{T}$ is equivalent to $\mathcal{L}$ as they receive a common sequence of messages on each process. Therefore, for any existing zero-buffer available equivalent class, the procedure above is able to find a representative zero-buffer legal history that strictly alternates sends and receives. □

Given the proof of message communication coverage, the soundness and completeness only rely on the existing proofs in the prior work. The soundness proof is consistent with the prior work: 1) the program order is precisely constrained in the encoding; and 2) any match pair used in a resolved satisfying schedule is valid. The deterministic receive operations and the collective operations for the new encoding do not violate these properties. To be precise, the first property still holds because the program order for deterministic receive operations and collective operations are precisely defined in the new encoding. The second property is also true because the set of match pairs is given as input, which is not affected by deterministic receive operations and collective operations.

The completeness proof of the new technique is similar to the prior work that uses the operational semantics to simulate the encoding during its operation to ensure that the two make identical conclusions. To prove the new encoding is complete, the operational semantics are extended to support deterministic receive operations and collective operations. A simulation of the extended operational semantics is then able to prove that any behavior of MPI semantics is encoded by the new technique in this paper. Because the soundness and completeness for the new encoding are both proved, we conclude that the encoding is correct for MPI semantics.

3.5 Experiment

We compare the performance of our work with two state-of-art MPI verifiers: ISP [49, 58], a dynamic analyzer, and MOPPER [20], a SAT based tool. We conduct a series of experiments for five typical benchmark programs that are modified to be single-path. Assertions related to correct computation are manually inserted into each program. All the results show the comparison between zero buffer semantics and infinite buffer semantics. The initial program trace for our approach is generated by running MPICH [39], a public implementation of the MPI standard. This program trace is encoded symbolically where each variable does not have a concrete value. A unique instance is generated for each write of a variable in the program

Table 3.1: Tests on Selected Benchmarks

Test Programs							Our Method		ISP		MOPPER	
Name	#Procs	#Calls	Match	B	Error	ZI	Mem	Time	#Runs	Time	Mem	Time
Monte	4	35	24	0	No	No[†]	3.62	0.02s	6	0.25s	6.09	<0.01s
				∞	No	–	3.42	0.02s	6	0.96s	–	–
	8	75	40K	0	No	No[†]	4.83	0.04s	>5K	TO	11.28	0.02s
				∞	No	–	4.34	0.04s	>5K	TO	–	–
	16	155	2E13	0	No	No[†]	8.97	0.29s	>5K	TO	24.42	0.08s
				∞	No	–	7.22	0.15s	>5K	TO	–	–
Integrate	8	36	5K	0	Yes	No	4.71	0.08s	1	0.15s	–	– [a]
				∞	Yes	–	4.20	0.04s	1	0.16s	–	–
	10	46	362K	0	Yes	No	5.39	0.08s	1	0.16s	–	– [a]
				∞	Yes	–	4.76	0.05s	1	0.26s	–	–
	16	76	1E12	0	Yes	No	8.79	0.62s	1	0.25s	–	– [a]
				∞	Yes	–	7.50	0.32s	1	0.54s	–	–
Diffusion2D	4	52	6E9	0	No	Yes	5.50	0.04s	90	3.09s	6.10	0.01s
				∞	No	–	4.80	0.03s	90	32.01s	–	–
	8	108	2E21	0	No	Yes	11.94	0.22s	>9K	TO	–	TO
				∞	No	–	8.51	0.12s	>9K	TO	–	–
	16	228	3E57	0	No	Yes	30.68	1.25s	>10K	TO	–	TO
				∞	No	–	30.76	5.11s	>10K	TO	–	–
Router	2	34	1	0	No	Yes	3.39	0.02s	1	0.04s	–	– [a]
				∞	No	–	3.37	0.02s	60	13.24s	–	–
	4	68	83K	0	No	Yes	4.18	0.02s	1	0.04s	–	–[a]
				∞	No	–	3.99	0.03s	>10K	TO	–	–
	8	136	7E9	0	No	Yes	5.17	0.04s	1	0.15s	–	–[a]
				∞	No	–	5.06	0.05s	>11K	TO	–	–
Floyd	8	120	4E29	0	No	No	13.87	0.15s	>20K	TO	18.05	0.27s
				∞	No	–	12.14	0.12s	>20K	TO	–	–
	16	256	1E58	0	No	No	21.58	0.26s	>20K	TO	67.53	43.08s
				∞	No	–	17.55	0.21s	>20K	TO	–	–
	32	528	3E137	0	No	No	252.97	439.89s	>20K	TO	212.30	476.52s
				∞	No	–	57.91	19.34s	>20K	TO	–	–

[†] MOPPER detects deadlock.
[a] MOPPER does not launch SAT analysis.

computation (similar to the static single assignment form [12]). Our encoding is resolved by the SMT solver Z3 [13]. MOPPER also needs an initial program trace with the same input data. MOPPER launches ISP to automatically generate such a trace. Since MOPPER is designed for deadlock checking, it does not encode any computation in a program. The results only show the performance of MOPPER for zero buffer incompatibility. The experiments are run on a AMD A8 Quad Core processor with 6 GB of memory running Ubuntu 14.04 LTS. We set a time limit of 30 minutes for each test. We abort the verification process if it does not complete within the time limit.

The results of the comparison are in Table 3.1. The column "Match" records the approximated number of match possibilities. A program with a large number of match possibilities has a large degree of message non-determinism. The column "ZI" indicates

whether the program is zero buffer incompatible or not. The column "Mem" records the memory cost in megabytes. The "Time" columns for our approach and MOPPER are only for constraint solving. As a note, our approach and MOPPER both spend less than one second to generate the trace and the encoding for every benchmark. The column "#Runs" for ISP is the number of program interleavings that ISP traverses before termination. The column "Time" for ISP is the running time of dynamic analysis. The meaning of the symbol "–" is "unavailable": either the test is not interesting for comparison or the error is detected in preprocessing.

Monte implements the Monte Carlo method to compute π [8]. It uses one manger process and multiple worker processes to send messages back and forth. In addition, barrier operations are used to synchronize the program.

Integrate uses heavy non-determinism in message communication to compute an integral of the sin function over the interval $[0, \pi]$ [1]. This benchmark also has a manger-worker pattern where the root process divides the interval to a certain number of tasks. It then distributes those tasks to multiple worker processes.

Diffusion2D has an interesting computation pattern that uses barriers to "partition" the message communication into several sections [1]. A message from a send can be only received in a common section.

Router is an algorithm to update routing tables for a set of nodes. Each node is in a ring and communicates only with immediate neighbors to update the tables. The program ends when all the routing tables are updated.

Floyd implements the Floyd's all-pairs shortest path algorithm [65]. Each node communicates only with the immediate following neighbor.

Note that all the tools are able to correctly check zero buffer incompatibility for the benchmarks above. Also, our encoding and ISP are both able to correctly check assertion violations. The results in Table 3.1 show that our encoding with Z3 is highly efficient compared to ISP. For the benchmark programs such as *Diffusion2D* and *Floyd*, where ISP

does not terminate after traversing a large number of interleavings, our approach returns under a second in most cases. Even for the benchmark programs where ISP terminates after traversing only a small subset of all the interleavings, our approach is able to run slightly faster. Our approach is also faster than MOPPER for the benchmark programs where there is a large degree of message non-determinism. If the number of match possibilities is low, our approach runs as fast as MOPPER does. As discussed earlier, a deadlock may be caused by many ways other than zero buffer incompatibility. The program *Monte* is zero buffer compatible, but it contains a deadlock that can be detected by MOPPER. Our solution was never intended to find such a deadlock. ISP should detect it but does not. For the programs *Integrate* and *Router*, MOPPER does not launch a SAT analysis because the ISP preprocessor detects the assertion violation or deadlock, and thus, MOPPER aborts the verification process.

3.6 Related Work

The dynamic analyzer ISP implements the POE algorithm, a Dynamic Partial Order Reduction (DPOR) algorithm [19] applied to MPI programs [58]. An extension is the MSPOE algorithm [49]. It operates by postponing the cooperative operations for message passing in transit until each process reaches a blocking call. It then determines the potential matches of send and receive operations in the runtime. In addition to program properties, it is able to check deadlocks.

Forejt et al. proposed a SAT based approach to detect deadlock in a single-path MPI program [20]. This solution is correct and efficient for programs with a low degree of message non-determinism. However, since the size of their encoding is cubic, checking large programs is time consuming. Similar to our solution, this work requires a match pair set that can be over-approximated.

MPI-Spin is integrated into the classic model checker SPIN [24] for verifying MPI programs [51, 52]. It generates a model of an MPI program and symbolically executes it. It does not scale to large programs with a large degree of message non-determinism.

Vo et al. used Lamport clocks to update auxiliary information via piggyback messages [62, 63]. While completeness is abandoned in their analysis, they show the approach is useful and efficient in practice.

Sharma et al. proposed the first push button model checker for MCAPI – MCC [48]. It indirectly controls the MCAPI runtime to verify MCAPI programs under zero buffer semantics. One drawback of this work is that it does not include the ability to analyze infinite buffer semantics which is known as a common runtime environment in message passing. A key insight, though, is the direct use of match pairs – couplings for potential sends and receives.

Elwakil et al. also used SMT techniques to reason about the program behavior in the MCAPI domain [17, 18]. State-based and order-based encoding techniques are both used. These techniques fail to reason about the infinite buffer semantics and require a precise match set which is non-trivial to compute beforehand.

Our prior work encodes an MCAPI execution into an SMT problem for detecting user-provided assertions [28]. The encoding is sound and complete and is easy to use to reason about infinite buffer semantics without requiring a precise match set. The work also provides an algorithm that runs in quadratic time complexity to generate a sufficiently small over-approximated match set based on the given execution trace.

3.7 Conclusion and Future Work

This paper presents a new algorithm that correctly encodes a single-path MPI program. This encoding, including the rules for MPI point-to-point communication and MPI collective communication, is capable of detecting zero buffer incompatibility. It is also adaptable to checking assertion violations. The key insight in this paper is that the new zero buffer

encoding considers only the schedules that strictly alternate sends and receives. This strict alternation strategy makes the encoding intuitive and easy to build automatically, and is able to cover all the message communication. Experiments indicates that our solution is correct and more efficient than two state-of-art MPI verifiers.

The encoding is dependent on a single-path MPI program which can be initialized by an execution trace. Future work will explore using bounded model checking to encode all the paths of an MPI program. This technique statically unrolls an MPI program and then verifies it by constraining the semantics into an SMT encoding. Also, future work will explore using the SMT encoding to check deadlock patterns other than zero buffer incompatibility.

Chapter 4

An Efficient Approach for Match Pair Approximation in Message Passing

Asynchronous message passing paradigm is commonly used in high performance computing (HPC). Message non-determinism makes the error detection in message passing programs very difficult. The prior work uses an over-approximation of the precise match pair records (each is a pair of a send and a receive that may potentially match in the runtime) to capture all possible message communication in a concurrent trace program (CTP). The SMT encoding with such a set of match pairs is able to witness program properties including deadlock, message race, and zero-buffer compatibility, but is inefficient because of the exponential ways of match pair resolution. This paper presents a new algorithm that under-approximates the match pairs for a CTP iteratively: first sectioning each process in the CTP such that each potential sender distributes roughly a bounded number of sends to match the same number of receives in the process, and then approximating the match pairs for the sends and receives in each section independently by a few simple rules with ranking. The algorithm runs in quadratic complexity in the number of operations. Novel in the work is that the algorithm has the flexibility to generate the match pair set with various size based on the user input. This paper further presents that the precise match pairs for any CTP can be generated with a bounded input. The experiments over a set of benchmarks show that the algorithm in this paper drastically reduce the runtime performance of property witnessing as all the properties are witnessed with a small set of match pairs generated by the new al-

gorithm. The results also show that the algorithm is able to scale to a program that employs a high degree of message non-determinism and/or a high degree of deep communication.

4.1 Introduction

Asynchronous message passing is a prevalent programming model in high performance computing (HPC). The idea of message passing is simple at the outset; processes communicate by sending messages from one to another. It does not take long, however, to realize that despite the simplicity of the programming model, there is a lot of subtlety in message passing that can impact program behavior. For example, depending on the library, a programmer must be aware of such things as message non-determinism (concurrent sends to a process can arrive in any order), buffering semantics in the runtime (a process may block if the buffer is full), or broad synchronization operations that message with groups of processes at the same time (i.e., collective operations). This inherent complexity means that the problems of determining if a message passing program is free of deadlock, compatible with a given buffering semantics, or if the message non-determinism affects the correctness of the computation are all NP-complete [20, 25, 26, 28]. Showing any of these properties for an input program requires a significant runtime for an approach.

Prior work on program correctness in the message passing paradigm can be roughly grouped into dynamic analysis where an existing runtime is manipulated to explore different scheduling outcomes [49, 58], model checking where a model of the original program is analyzed [51, 52], runtime verification where the program execution is observed but not manipulated [23, 32, 60], and symbolic model checking where a model of the program is symbolically analyzed with an SMT solver [25, 28]. This paper looks specifically at symbolic model checking and ways to better leverage the SMT solver in witnessing program properties for message passing programs.

The most efficient types of encoding for symbolic model checking of message passing programs, where efficient means they scale to programs with a high degree of message non-

determinism, rely on the concept of a *match pair* and a *concurrent trace program* abstraction (CTP) [17, 18, 20, 25, 26, 28]. A match pair is a send and receive pair that may be matched by the run-time in some feasible execution of the program. A CTP is an abstraction of the program that removes all branching so each process is a sequence of send, receive, and wait (e.g., a witness to the completion of a send or receive) operations [25, 28]. The abstraction is constructed by observing the execution of the program in the run time for a given input. Symbolic model checking gives the SMT solver an encoded CTP with a set of match pairs, and the solver then tries to select a subset of the given match pairs in such a way that the CTP deadlocks or violates an assertion.

The size of the input match-pair set directly impacts the cost of the call to the SMT solver in the symbolic model checking of message passing programs; a large input set typically means an expensive solver call especially if there is a high-degree of message non-determinism. The idea in this paper is to reduce the cost of the solver call by reducing the size of the input match set to each call. Such a reduction cannot, however, be done naively; any match pair set must be *message complete*. A message complete match pair set for a CTP is one that lets the program run to completion (assuming it is free of deadlock).

To reduce the number of match pairs and thereby decrease the cost of verification, this paper describes an iterative algorithm to successively generate under-approximations of the true set of match pairs until all match pairs are generated. The under-approximations retain the message complete property. Such an approach focuses on property witnessing rather than proofs of correctness as the intent is to witness program properties early in the iterations before the set of match pairs gets large.

Generally, the algorithm generates the match pairs in two steps. First, each process is sectioned, where a section contains a sequence of receives in the process and a number of sends from all the senders that may potentially match these receives. The number of sends from each sender is roughly bounded by k given as a user input. The sends from each sender are distributed sequentially. The total number of the sends distributed to the section is equal

to the number of receives in the same section. Match pairs are then generated for each section independently. This independent sectioning effectively ignores combinations of match pairs, that are feasible, but only available when the considered concurrent outstanding sends are more than the number of receives in the section. The algorithm generates the match pairs for the sends and receives in each section by a list of simple rules based on ranks [28]. A rank is a non-negative integer that represents the position of a send or a receive in a specific sequence in the section.

Given the algorithm for match pair generation, the approach in this paper starts with $k = 1$ and increments k to a larger bound such that the number of the divided sections is changed, because the match pair generation for the new iteration is only meaningful if each process is sectioned differently. The growth of the size of the match pair set is polynomial as k increases. The program property is then witnessed with the larger bound k; or the process repeats.

The paper includes the discussion that the precise match pairs for any CTP can be generated with a sufficiently large bound k. Experiments further show that the runtime cost of witnessing properties is dramatically reduced as the properties are witnessed with relatively small values of k creating simple problem instances for the SMT solver. Experiments also show that the algorithm is able to scale to a program that employs a high degree of message non-determinism and/or a high degree of deep communication. Deep communication in this context means that each sender for a receiver issues a long sequence of sends to the receiver. The contributions include,

- the efficient algorithm that under-approximates the precise match pairs for a CTP,
- the proof that the precise match pairs for any CTP can be generated by the new algorithm with a sufficiently large bound k, and
- the results from the experiments using a set of benchmarks to demonstrate the improved ability of witnessing program properties.

p_1	p_2	p_3
$r_0(from\ *,\ to\ p_1,\ w_0)$	$s_0(from\ p_2,\ to\ p_1)$	$s_3(from\ p_3,\ to\ p_1)$
$w_0(r_0)$	$s_1(from\ p_2,\ to\ p_1)$	$s_4(from\ p_3,\ to\ p_1)$
$r_1(from\ *,\ to\ p_1,\ w_1)$	$r_5(from\ p_1,\ to\ p_2,\ w_5)$	
$w_1(r_1)$	$w_5(r_5)$	
$r_2(from\ p_2,\ to\ p_1,\ w_2)$	$s_2(from\ p_2,\ to\ p_1)$	
$w_2(r_2)$		
$s_5(from\ p_1,\ to\ p_2)$		
$r_3(from\ *,\ to\ p_1,\ w_3)$		
$w_3(r_3)$		
$r_4(from\ *,\ to\ p_1,\ w_4)$		
$w_4(r_4)$		

Figure 4.1: A simple concurrent trace program.

The rest of the paper is organized as follows: Sections 2 presents the definition and semantics of CTP; Section 3 presents the general algorithm in the paper; Section 4 states that the precise match pairs can be generated; Section 5 gives the experimental results; Section 6 discusses the related work; and Section 7 is the conclusion and future work.

4.2 Concurrent Trace Program Definition and Semantics

This section explains message communication in a simple CTP consisting of a handful of operations. Consider the CTP in Figure 4.1 that includes three processes that use non-blocking send (s) and non-blocking receive (r) for message communication. The nearest-enclosing wait (w) witnesses the completion of the receive [28]. The completion of any send or receive, is only confirmed when the send or the receive is matched in the runtime. Note that if the sender ID for a receive is "$*$", then the receive is wildcard meaning that it may match a send from any sender.

Picking up the scenario in Figure 4.1, process p_1 receives five messages from any sender or the specific senders and sends one message to p_2; process p_2 sends three messages to p_1 and receives a message from p_1; and process p_3 sends two messages to p_1.

A set of match pairs represents a solution to the SMT problem if that set of match pairs is feasible. Feasible in this context means there exists a program trace allowed by the runtime that matches according to the solution set. In this example, a feasible program trace is equation (4.1).

$$\begin{aligned} s_0 \to r_0 \to w_0\langle r_0\ s_0\rangle \to s_3 \to r_1 \to w_1\langle r_1\ s_3\rangle \to s_1 \to r_2 \to w_2\langle r_2\ s_1\rangle \\ \to s_5 \to r_5 \to w_5\langle r_5\ s_5\rangle \to s_4 \to r_3 \to w_3\langle r_3\ s_4\rangle \to s_2 \to r_4 \to w_4\langle r_4\ s_2\rangle \end{aligned} \tag{4.1}$$

As shown, the feasible trace in equation (4.1) implies a set of match pairs that represents a solution to the SMT problem from the prior work. The solution for equation (4.1) is $\{\langle r_0\ s_0\rangle, \langle r_1\ s_3\rangle, \langle r_2\ s_1\rangle, \langle r_5\ s_5\rangle, \langle r_3\ s_4\rangle, \langle r_4\ s_2\rangle\}$.

Since the message matching is possibility non-deterministic in the presence of wildcard receives, there exist other feasible executions for the CTP in Figure 4.1 where the receives are matched in different ways. For example, the receive r_0 can be matched with the send s_3 instead of the send s_0 if s_3 arrives in p_1 earlier than s_0. As such, the message delivery for r_0 is non-deterministic and it is associated with two potential match pairs. Given the message non-determinism, the message communication can be resolved in many (and possibly exponential) ways. Therefore, to capture the behavior in other executions, new match pairs are needed. For example, another feasible execution is presented in equation (4.2). As shown, the CTP in Figure 4.1 deadlocks for this trace.

$$s_0 \to r_0 \to w_0\langle r_0\ s_0\rangle \to s_1 \to r_1 \to w_1\langle r_1\ s_1\rangle \to s_3 \to s_4 \to \text{(Deadlock)} \tag{4.2}$$

The deadlock occurs because there is no way to match the receives r_2 and r_5 after the sends s_3 and s_4 are issued. The use of the new match pair $\langle r_1\ s_1\rangle$ is essential to find the execution

in equation (4.2) that makes the CTP in Figure 4.1 deadlock. The goal of the new algorithm in this paper is to generate a reduced set of the precise match pairs that can be used to finding such a trace with hidden errors.

Given the example in Figure 4.1, the algorithm in this paper would initialize $k = 1$, meaning that each sender has roughly one send distributed to a section. The algorithm sections process p_1 with $k = 1$ in equation (4.3).

$$\textit{section 1} :\{r_0, r_1, s_0, s_3\},\ \textit{section 2} :\{r_2, r_3, s_1, s_4\},\ \textit{section 3} :\{r_4, s_2\} \tag{4.3}$$

Since there are two senders for p_1, each sender distributes one send to a section to match the same number of receives. For example, *section 1* in equation (4.1) contains the receives r_0 and r_1, and the sends s_0 and s_3. The algorithm in this paper then sections process p_2 in the same way. Finally, the match pairs for each section are over-approximated using ranks [28]. Intuitively, a rank is the position of a send or a receive in a list. For example, r_0 has the rank of "0" and r_1 has the rank of "1" in the receive list for *section 1*. Given the ranks, the match pairs $\langle r_0\ s_0\rangle$, $\langle r_0\ s_3\rangle$, $\langle r_1\ s_0\rangle$, and $\langle r_1\ s_3\rangle$ are generated for *section 1*. The generated match pairs with $k = 1$ are sufficient to capture the match solution for the trace in equation (4.1). Since the match pair $\langle r_1\ s_1\rangle$ is not generated with $k = 1$, the trace in equation (4.2) is not captured.

The algorithm then increments k to 2 to repeat match pair generation.

$$\textit{section 1} :\{r_0, r_1, r_2, r_3, s_0, s_1, s_3, s_4\},\ \textit{section 2} :\{r_4, s_2\} \tag{4.4}$$

As shown in equation (4.4), the algorithm with $k = 2$ is able to partition process p_1 into two sections, and *section 1* groups the receive r_1 and the send s_1 at present. As such, the match pair $\langle r_1\ s_1\rangle$ is generated and the deadlock in equation (4.2) is witnessed.

4.3 Main Algorithm

This section presents how to generate match pairs for a CTP. Intuitively, Algorithm 1 is called several times for the CTP, once for each process that contains a sequence of receives. In each call of Algorithm 1, a process is sectioned, where each section contains roughly k sends from each sender. A sequence of receives in the process is also added to the section where the number of receives is equal to the number of sends in the same section. The divided sections consisting of only sends and receives for the process are then input to the existing algorithm for match pair generation [28].

Algorithm 1 Process Sectioning

```
1: while |R| > 0 do
2:     r ← DEQUEUE(R)
3:     if FRM(r)= * then
4:         let p be such that ∀p' ∈ Δ (|S(p)| > 0 ∧ |S(p')| > 0 ∧ N_s(p) ≤ N_s(p'))
5:     else
6:         let p be the sender of r
7:     end if
8:     N_s(p) ← N_s(p) + 1
9:     s ← DEQUEUE(S(p))
10:    cur ← cur ∪ {r, s}
11:    if ∀p ∈ Δ (N_s(p) >= k ∨ |S(p)| = 0) ∨ |R| = 0 then
12:        secs ← secs ∪ {cur}
13:        cur ← ∅
14:        for p ∈ Δ do
15:            N_s(p) ← 0
16:        end for
17:    end if
18: end while
```

4.3.1 Process Sectioning

At a low level, the presentation needs to first explain a few data structures that are essential to Algorithm 1. R is a list of all the sequential receives in a process. Δ is a set of IDs of all the senders $\{p_1, p_2, \ldots\}$. The current section *cur* represents a new program that contains

only sends and receives. It is initialized to an empty set. The set *secs* is used to store all the partitioned sections.

The algorithm also applies a few auxiliary functions. S(p) returns a list of sends for the sender p. $N_S(p)$ returns the count of sends directed to the current section by the sender p. DEQUEUE(L) removes and returns the first element in the list L. FRM(r) returns the sender ID of the receive r.

Given the data structures and functions, Algorithm 1 runs in several steps:

1. Dequeuing the first receive r from the list R (line 2);
2. Choosing a sender p based on the sender ID of r (line 3 to line 7);
3. Incrementing the count $N_s(p)$ and dequeuing the first send s from the send list of p (line 8 to line 9);
4. Adding the combination of r and s to the current section *cur* (line 10);
5. Adding the current section *cur* to the set *secs* and then resetting *cur* and N_s once $N_s(p)$ is greater or equal to k for each sender $p \in \Delta$ or there are no sends in $S(p)$ or there are no receives in the destination process (line 11 to line 17);
6. Repeating the above steps until R is empty (line 1).

For each receive r, if the sender ID is $*$ indicating r is wildcard, then the algorithm randomly chooses a sender p that has the minimum count among the senders where each sender has at least one send at line 4. If r is a deterministic receive, then p is the specific sender for r at line 6.

The intuition of the condition at line 11 is that each sender must distribute roughly k sends to the current section unless the receive list R is already empty. The sender may distribute more than k sends as if more than k deterministic receives that specify the sender are added to the section. If the total number of sends in the sender is less than k, then all the sends must be distributed from that sender.

4.3.2 Match Pair Approximation

Given the partitioned sections in the set *secs* by Algorithm 1, the match pairs for each section can be generated independently by the existing algorithm in the prior work [28].

Intuitively, the algorithm checks all the pairs of receives and sends in each section, and prunes obvious matches that cannot exist in any runtime implementation of the specification by comparing ranks of sends and receives. A rank of a receive is its position in the list R. A rank of a send is its position in the send list $S(p)$ from the sender p to the receiver.

The existing algorithm implies the "match over-approximation" property, in that all the precise match pairs (and maybe some unprecise match pairs) in the section are generated. Please refer to the prior work [28] for more detail of the algorithm.

4.4 Coverage of Message Communication

According to Algorithm 1, a send and a receive from two different sections are not considered for matching. For the CTP in Figure 4.1, the match pair $\langle r_1\ s_1 \rangle$ can not be generated with $k = 1$ as the receive r_1 and the send s_1 are partitioned into two sections. To capture the missed match pairs for a send and a receive, a possible way is to group them into a single section with a larger bound k. The match pair $\langle r_1\ s_1 \rangle$ for the example in Figure 4.1 can be generated with $k \geq 2$. Note that all the precise match pairs for the CTP in Figure 4.1 are generated with $k = 3$, where each process is partitioned into a single section.

Definition 28 (*Max Bound*)**.** *For any CTP, there exists a bound k, such that each process can be partitioned into a single section; this bound k is called the max bound.*

The *max bound* of k can be statically computed. According to Algorithm 1, a section can be constructed only if each sender reaches the k-bound for send distribution or distributes all the sends assuming the receive list is not empty. As such, it implies that

$$k = \text{MAX}(\{|S(p_i)| \mid p_i \in \mathit{sender}\}), \tag{4.5}$$

that is able to group all the sends to a single section in the destination process, where the function MAX returns the maximum among a set of numbers. It is assumed that the total number of receives in a process is equal to the total number of sends directed to the process. Therefore, the process is partitioned into a single section. Further, if the bound k satisfies that

$$k = \text{MAX}(\{k_j \mid p_j \in P\}), \tag{4.6}$$

where P is a set of all the processes in the CTP and k_j is a bound that satisfies equation (4.5) for process p_j, then each process in the CTP is partitioned into a single section. The bound k that satisfies equation (4.6) is the *max bound*.

According to the *max bound* of k and the match over-approximation property of the existing algorithm [28], the precise match pairs for a CTP can be over-approximated by the algorithm in this paper.

4.5 Experiments

This section describes a series of experiments over a set of benchmarks to determine (1) the effectiveness of the algorithm is quickly witnessing properties of the input program, and (2) how well the algorithm controls the size of the match pair set as the bound k increases.

The benchmark programs are specific to the Message Passing Interface (MPI), a widely used message passing standard [3] and come from different sources. Three are derived from actual MPI programs and are used in other works for benchmarking [1, 40, 65].

- *Diffu2DNoBa* is modified from the program *Diffusion 2D*, which uses barriers to partition the message communication into several sections [1]. *Diffu2DNoBa* removes the barriers from the original program so deadlocks are present in the new program. The program is also interesting because the messages to any receiver are distributed from a large set of senders.

- *Pktuse* is executed with only 5 processes – each of which has a long sequence of messages to be sent to the other processes [40]. The program uses wildcard receives only, therefore has a high degree of message non-determinism.
- *Floyd* implements the shortest path algorithm for all the pairs of nodes [65]. Each node communicates only with the immediate following neighbor. The program also has a large set of senders for each receiver.

The remaining four are synthetic programs created for this study.

- *DeepComm* is a simple program with one receiver and 4 senders. The program is designed to have a long sequence of sends in each sender. Also, this scenario issues only wildcard receives, so that the messages from different senders may race.
- *MultiM* is an extension to a program in the MCAPI library distribution [28]. The program adds extra iterations to the original program to generate longer execution trace. The program uses only wildcard receives and is designed to have an interesting violation of assertion that only occurs in some possible executions.
- *Mismatch* is designed to contain a communication deadlock in execution. The program interleaves wildcard receives and deterministic receives in the program text. A deterministic receive may be orphaned in program execution leading to a deadlock as all the potential sends it need are matched with the preceding wildcard receives.
- *MismatchEx* is an extension to the program in Figure 4.1 that contains more sends and receives. Similar to the program in Figure 4.1, a deadlock may occur in deep execution.

These programs are tested for three types of properties: assertion violation, zero buffer compatibility and deadlock. The deadlock checking relies on a static pattern matcher to identify potential deadlock scenarios [26]. The experiments here use single pattern match, and the SMT encoding is to witness the feasibility of that match. The same pattern match is used for all experiments on any given deadlock example. The initial trace for any input program is generated by MPICH [39], a public implementation of the MPI standard. The SMT encoding for each test is generated by the existing rules [25, 28] and is solved by Z3

Table 4.1: Tests on Selected Benchmarks

	Test Programs			Performance			
	Name	#P	#Calls	k	#Match	Time	Speed
AssertV	*Diff2DNoBa*[d]	16	188	3	1,066	87.361s	0.59
	DeepComm	5	180	1	240	0.837s	1,647
	Pktuse	5	2048	1	1,792	180s	>40
	MultiM	3	266	1	500	10.892s	295
	Floyd[d]	32	528	2	1,928	60.546s	0.48
ZeroCom	*Diff2DNoBa*[d]	16	188	3	1,066	10.177s	0.53
	DeepComm	5	180	1	240	0.875s	291
	Pktuse	5	2048	1	1,792	121s	>59
	MultiM	3	266	1	500	8.312s	366
	Mismatch	3	800	1	204	2.904s	3.06
	MismatchEx	3	296	1	165	0.579s	876
	Floyd	32	528	1	1,928	89.032s	1.02
DL	*Diff2DNoBa*	16	188	1	514	3.479s	13.50
	Mismatch	3	800	1	204	2.160s	6.61
	MismatchEx	3	296	3	328	1.253s	27.23

[d] The property does not exist for the benchmark.

[13]. The experiments are run on a AMD A8 Quad Core processor with 6 GB of memory running Ubuntu 14.04 LTS. A time limit of 2 hours is set for each test. The test aborts the verification process if it does not complete within the time limit.

4.5.1 Effectiveness

The *effectiveness of property witnessing* are shown in Table 4.1 that divides the tests into three groups, where each group is the tests for a single type of property. Each group is labeled in the first column: "AssertionV" means assertion violation; "ZeroCom" means zero buffer compatibility; and "DL" means deadlock. For each benchmark, k is the minimum value at which the property is witnessed (if the property exists), or the value at which the over-approximated match set is generated (if the property does not exist). The column "#M" is the number of the generated match pairs for k. The column "Time$^*_k$" is the cumulative sum: $\sum_{i=1}^{k} \text{Time}_i$, where Time$_i$ is the runtime of checking satisfiability with the generated match pairs for the bound i. The column "Speed" reports the relative speedup over the running time of simply using the regular over-approximated match pair set: Time(*over-approximated*)/Time$^*_k$.

The results show that each property, if existing for a benchmark, is efficiently witnessed by the SMT encoding with the under-approximated match pairs in the benchmark; it is not known if such an early detection takes place in general. The tested properties for all the real programs can be witnessed with $k = 1$. For example, the assertion violation for the program *Pktuse* can be detected with $k = 1$ where the speedup is greater than 40. Note that the speedup for the program *Pktuse* is not a concrete number because the test with the over-approximated match pairs is time out.

For the tests where the properties do not exist in the benchmarks, the results show slow down as expected, but the intent of this work is to improve the ability to find witnesses early rather than to prove correctness. For example, three tests are run for checking the assertion violation property for the program *Diffu2DNoBa*, where k is incremented from 1 to 3. The over-approximated match pairs are generated with $k = 3$ that is the *max bound* and can be easily computed.

The tests for all the synthetic programs also show the properties can be witnessed with a potential for speed up, even if several tests are required to be run beforehand. For example, the program *MismatchEx* is designed to contain a deadlock deep in an execution. The witness is found after k is incremented to 3, where the speed up is more than 27 compared to the runtime with the over-approximated match set for $k = 42$. The capability of witnessing properties with a low k-bound is especially helpful for the large programs to be feasible as the over-approximated match set for such programs is usually too large to be resolved.

4.5.2 Scalability

To evaluate how the input k impacts the generated match pairs, the presentation discusses two benchmarks for how the sends are distributed from senders. The benchmarks are outside the set of programs discussed earlier and are defined based on the template program in Figure 4.2. The template program is simplified to contain only a receiver p_r that issues a

p_r	p_1	...	p_m
$r_0(from\ *,\ to\ p_r,\ w_0)$	$s_0(from\ p_1,\ to\ p_r)$	...	$s_{m0}(from\ p_m,\ to\ p_r)$
$w_0(r_0)$	$s_1(from\ p_1,\ to\ p_r)$		$s_{m1}(from\ p_m,\ to\ p_r)$
$r_1(from\ *,\ to\ p_r,\ w_1)$	...		...
$w_1(r_1)$			
...			

Figure 4.2: A template concurrent trace prorgam.

sequence of receives, and several senders $p_1, \ldots, p_m$ where each sender issues a fixed number ($|S|$) of sends. The number of receives in p_r is $|S| \times m$.

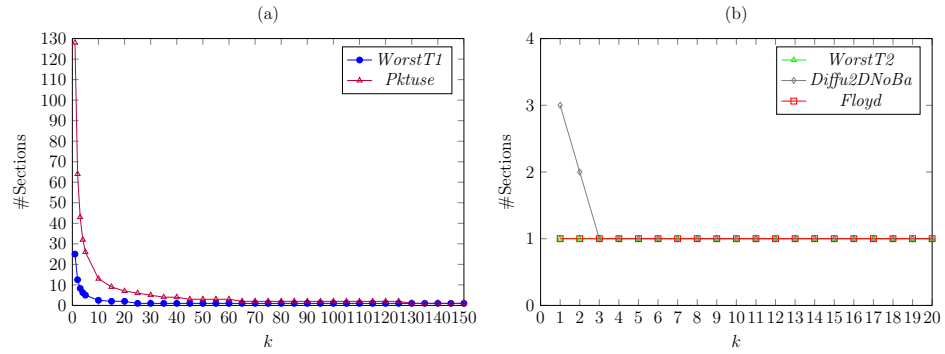

Figure 4.3: The number of partitioned sections of varying the input k.

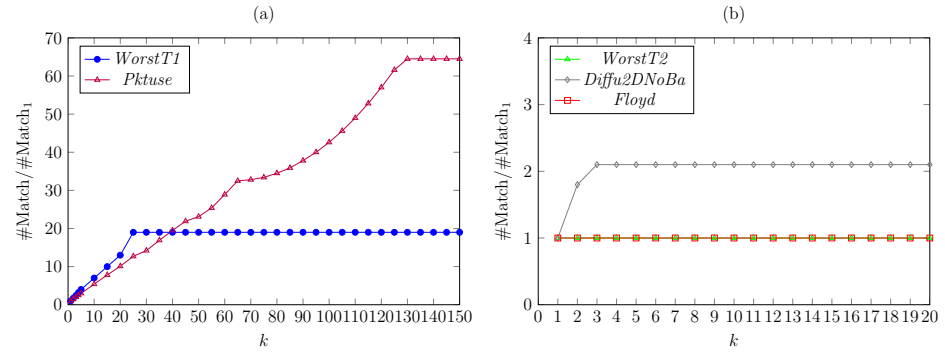

Figure 4.4: The relative growth of the number of match pairs of varying the input k, where Match_1 is the number of the generated match pairs for $k = 1$.

The first benchmark, *WorstT1*, is constructed by restricting all the receives in Figure 4.2 to be wildcard, which implies the worst case of message non-determinism. Also, let

$m = 4$ and $|S| = 25$ in the program configuration so there are more than one sections partitioned for a set of possible k-bounds. In contrast, the best case of message non-determinism is a program that only contains deterministic receives. As such, the matches for the best case program are deterministic.

The second benchmark, *WorstT2*, is related to the worst case of wide communication. Wide communication in this context means that the sender only sends one message to the receiver. The program is constructed by setting $|S| = 1$ and $m = 100$. As for the best case of wide communication, the program has to contain only the deep communication where a receiver has exactly one sender. The matches for this type of best case are deterministic because the messages are received in a fixed FIFO order.

Given the two worst-case programs, the presentation discusses how the number of sections and the number of match pairs grow as k increases for the two programs. The relationship between the number of sections and k is illustrated in Figure 4.3, and the relationship between the number of match pairs and k is illustrated in Figure 4.4. Note that the values of y-axis in Figure 4.4 are computed by dividing the number of match pairs for any k by the number of match pairs for $k = 1$.

Figure 4.3 (a) shows that the growth of the number of sections is not linear for the program *WorstT1*. As such, incrementing k may not be meaningful if the number of sections does not change. The approach in this paper only increments k until the number of sections changes, and then outputs the generated match pairs with k to the SMT encoding. Since the number of the generated match pairs per section is also non-linear according to the complexity of the existing algorithm [28], the total number of the match pairs for *WorstT1* in Figure 4.4 (a) has a sharp growth between small values of k, and then grows more gently until it reaches the bound, which represents the over-approximated match pairs. This observation demonstrates that the algorithm scales for a small range of k where the generated match set is small.

For the program *WorstT2*, Figure 4.3 (b) shows that the number of sections is always one as expected because only one section can be partitioned for wide communication. As such, the total number of match pairs for *WorstT2* in Figure 4.4 (b) does not change.

Based on the discussion of message non-determinism and wide communication, the synthetic programs discussed earlier can be grouped. The programs *DeepComm* and *MultiM* are close to the worst case of message non-determinism as only wildcard receives are employed in both programs. For the wide communication, no synthetic program is close to the worst case. Instead, all the programs show a certain degree of deep communication.

The real programs can also be classified for message non-determinism and wide communication. Figure 4.3 and Figure 4.4 further plot the growths for the real programs. The programs *Diff2DNoBa* and *Floyd* uses more wide communication, therefore, their growths in Figure 4.4 rapidly reach the bound of match pairs. In contrast, the program *Pktuse* employs a large degree of deep communication with many wildcard receives. As such, *Pktuse* in Figure 4.4 (a) gradually grows to the bound as k increases to 128. Thefore, the presentation demonstrates that the algorithm is able to scale to a program with a high degree of message non-determinism and a high degree of deep communication.

4.6 Related Works

There are several works related to match pairs. Sharma et al. proposed the first push button model checker for MCAPI – MCC [48]. It indirectly controls the MCAPI runtime to verify MCAPI programs under zero buffer semantics. An obvious drawback of the work is its inability to analyze infinite buffer semantics which is known as a common runtime environment in message passing. A key insight, though, is the direct use of match pairs.

A precise SMT encoding technique is proposed for detecting user-provided assertions for MCAPI programs [28]. The encoding is sound and complete and is easy to use to reason about infinite buffer semantics without requiring a precise match set. The work also provides an algorithm that runs in quadratic time complexity to generate a over-approximated match

set based on the given execution trace. This approach is extended to checking zero buffer incompatibility for MPI semantics [25].

Elwakil et al. proposed another encoding technique that is applied to MCAPI [17, 18]. The encoding uses a non-obvious way to constrain the happens before relation in a program. The approach fails for two reasons. First, it does not find out all possible process interleavings under infinite buffering setting. Second, the foreknowledge assumption about the potential matches of send and receive operations does not apply for a large complex program execution.

Forejt et al. proposed a SAT based approach to detect deadlock in a single-path MPI program [20]. The solution is correct and efficient for programs with a low degree of message non-determinism. However, since the size of the encoding is cubic, checking large programs is time consuming. The work also requires a match pair set.

There are other solutions for message passing program analysis. The dynamic analyzer ISP implements the POE algorithm, a Dynamic Partial Order Reduction (DPOR) algorithm [19] applied to MPI programs [58]. An extension is the MSPOE algorithm [49]. It operates by postponing the cooperative operations for message passing in transit until each process reaches a blocking call, and then determines the potential matches of send and receive operations in the runtime. A drawback of ISP is that it does not scale for large programs due to state explosion.

Umpire applies runtime verification for MPI programs [60]. The approach takes one manger thread and several outfielder threads in an MPI execution. A drawback of the approach is that it relies on a concrete execution, which may miss the errors in the other execution trace. The extensions to Umpire is Marmot [32] and MUST [23]. These approaches are neither sound nor complete for deadlock detection.

MPI-Spin is integrated in the model checker SPIN [24], for verifying MPI programs [51, 52]. It generates a model of an MPI program and symbolically executes it. It does not scale to large programs with a large degree of message non-determinism.

CIVL is a model checker which uses symbolic execution to verify a number of safety properties of various types of concurrent programs including message passing programs [56, 67].

Vo et al. proposed an approach that uses Lamport clocks to update the auxiliary information via piggyback messages [62, 63]. While completeness is abandoned in their analysis, the work is useful and efficient in practice.

4.7 Conclusion and Future Work

This paper presents a new algorithm that generates the match pairs for message passing programs in the context of a CTP. First, the algorithm sections each process where each section contains roughly k sends from each sender that may match the same number of receives in the section. The bound k is a user input. The algorithm then approximates the match pairs for each section [28]. The key insight of the algorithm in this paper is that the match pairs for each section are generated independently. This paper presents that all the precise match pairs for a CTP can be generated with the *max bound* of k. Experiments demonstrate that all the properties in the benchmarks can be efficiently witnessed with a low k-bound. Experiments also show that the algorithm scales to a program that employs a high degree of message non-determinism and/or a high degree of deep communication.

The algorithm in this paper is restricted to the CTP abstraction that only reveals the behavior from an execution trace given a concrete input. The program structures that do not exist in a CTP but are commonly used in any message passing program, such as branches, loops, functions, are not supported by the algorithm in this paper. Future work will explore to extend the algorithm in this paper to support these structures. As discussed earlier, the algorithm in this paper does not scale well to a program with wide communication due to how a process is sectioned. Future work will consider new ways to section processes so that programs with various communication types are supported.

Chapter 5

A Hybrid Approach of Dynamic and Static Analyses for Deadlock in MPI Programs

The message passing interface (MPI) is a common programming model for distributed computing. Message race (intended or not) leads to non-determinism making test and debug in MPI programs very difficult. This paper addresses the NP–complete problem of deadlock detection in MPI programs in the context of trace programs. The initial trace programs are preprocessed by executing the program. The solution uses progressively more precise analyses to generate and then prune a potential set of deadlocks: static matching to identify deadlock pattern instances; execution of the instances on an abstract machine to reject those that are provably non-feasible; and finally, if needed, validation of the instances to remove any remaining that are non-feasible. Novel in the work is the abstract machine based on counting to efficiently reject many non-feasible instances without exhaustively enumerating all message races. The paper further defines two deadlock patterns: circular dependency and orphaned receive. The first pattern relies on simple rules for validation, while the second requires a higher cost SMT encoding. The paper proves the approach sound and complete for each pattern and compares the approach with two other deadlock tools on typical benchmarks. The comparison shows that the new approach scales in the presence of millions of possible send-receive matches and completes on benchmarks where the other tools time out. The experiments also show that the two deadlock patterns in the paper cover all the deadlock cases in benchmarks.

5.1 Introduction

The Message Passing Interface (MPI) is widely used in high performance computing (HPC). There are many features supported by MPI 3.1 standard [3]. In a brief summary, one-sided communication allows remote memory access (RMA) in MPI programs. MPI provides message passing in a point-to-point way or in a collective way. Hybrid programming with threads uses multi-threads within a process. Hybrid programming with shared memory allows a process to allocate shared memory through MPI. Also, MPI can create topology for processes and connects these processes by neighborhood collectives.

A problem endemic to concurrent programming is deadlock. In the context of this paper deadlock is defined as "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" [41]. Determining whether an MPI program deadlocks is NP–Complete [20]. The possible state-space of a program scales exponentially with respect to the number of processes executing. Message races, intended or not, can cause non-deterministic behavior that expands the state-space even further. These problems compounded with the intricacies of MPI semantics make it incredibly difficult to debug deadlock, even in trivial programs.

There are a few classes of algorithms proposed for deadlock detection. One class focuses on enumerating the pairs of sends and receives that may potentially match at runtime. The number of match pairs explodes as more sends and receives are added to the program, which leads these analyses to scale poorly [20, 49, 58]. Another class of algorithms start with a set of potential deadlocks that can be efficiently detected, and then to predict real deadlocks from the set using static analysis. These strategies generally scale much better on large programs [30, 46, 50]. The general algorithm presented here is inspired by the latter.

This paper targets the central facet of the MPI standard: point-to-point communication. The paper presents a simple set of communication primitives to semantically model the MPI point-to-point communication procedures. Memory allocation, access, and datatypes are not within the scope of this presentation as it is specifically focused on messaging dead-

lock. Strategies for expanding these analyses to other portions of the MPI standard are discussed in section 5.9. The approach in this paper applies to MPI programs that do not decode data, meaning that they do not employ data dependent control flows, and do not alter their control flows based on which sends a non-deterministic receive matches with. It is a reasonable assumption since a large class of programs applied to HPC are coded in a manner that is consistent with the simplification. The approach in this paper then detects deadlocks in the context of trace programs.

This paper presents a new algorithm that is able to detect feasible deadlock for an MPI program trace in three steps. Before analysis the input trace must be generated either by symbolic execution or running the program under instrumentation. The algorithm first detects the set of deadlock pattern instances by statically traversing the program's trace. It then uses an abstract machine to prune provably non-feasible pattern instances from the set of potential deadlocks. Finally, the algorithm validates whether or not any of the remaining instances imply a real deadlock by means of an SMT encoding. The approach in this paper is also adaptable to lock based concurrent programs.

Novel in this paper is an abstract machine that efficiently rejects non-feasible instances by counting the issued sends and receives rather than exhaustively enumerating all message races. The complexity of the machine is quadratic with respect to the number of communications.

This paper also defines two distinct patterns of deadlock with their validation methods: circular dependency and orphaned receive. A circular dependency exists when there is a cycle among a group of processes where a receive on each member process waits for the issuing of a send on another member process but never gets a response. This causes the program to deadlock since there is no ordering to resolve the first program's dependence on itself. An orphaned receive pattern consists of a pair of receives on the same process: a *wildcard* (a receive that may match a send from any source) and a *deterministic* receive (a

receive that only matches a send from a indicated source). The deadlock occurs when the runtime matches the wildcard receive to the send needed by the deterministic receive.

This paper presents how to detect all the instances for each of the patterns above by statically traversing the program. We also present how to validate the instances of the circular dependency pattern by comparing the counts of issued operations, and how to validate instances of the orphaned receive pattern that requires a higher cost SMT encoding. The presentation first gives the pattern match algorithm and the validation algorithm for infinite buffer semantics. The modification to zero buffer semantics is discussed latter.

The paper includes proofs that the general algorithm is sound and complete for the patterns of circular dependency and orphaned receive. Also, the comparison with two state-of-art MPI verifiers shows that the new approach scales to large benchmark programs while other tools time out. The experiments further show that the two deadlock patterns in this presentation cover all the deadlock cases in benchmark programs.

The contributions are summarized as follows.

- The primary contribution is an abstract machine that is able to efficiently prune provably non-feasible pattern instances from a set of potential deadlocks;
- The second contribution is the definition of two typical patterns: circular dependency and orphaned receive, and their validations; and
- The third contribution is the implementation of the algorithm with a comparison to two state-of-art MPI verifiers over a set of benchmark programs. The comparison shows that the new approach scales to large benchmark programs while other tools time out.

The rest of the paper is organized as follows: Section 5.2 introduces the concurrent trace program for an MPI program; Section 5.3 presents the general algorithm in this presentation, Section 5.4 and 5.5 present the circular dependency pattern and the orphaned receive pattern with their pattern matching algorithms and their validation algorithms; Section 5.7

gives the experimental results; Section 5.8 discusses the related work; and Section 5.9 is the conclusion and future work.

5.2 Concurrent Trace Programs

The presentation only considers non-determinism arising from message races and defines MPI program semantics in the context of concurrent trace programs (CTP) [64]. The CTP is initialized by an execution trace generated by the runtime. The syntax is in Figure 5.1. Lists use white space rather than commas to delineate members, ellipses (...) represent zero or more repetitions, and bold-face font denotes terminals in the language.

5.2.1 Definition

A CTP (*ctp*) is a list of processes. A process (p) is a list of commands. The commands (e) are: non-blocking send (**s**), non-blocking receive (**r**), wait (**w**), suspended wait ($*$**w**), barrier (**b**), and counted barrier ($*$**b**). The non-terminal x in the grammar is a unique string identifier **ID** associated with a receive, a send, a wait, or a suspended wait. The non-terminals v and y in the grammar are used to denote endpoints.

The receive and send commands include source and destination endpoints for communication. In the case of receive, the source may be $*$ indicating that the receive can match with any send to the same destination regardless of the source. The $*$ is only valid as the source of a receive and is not used anywhere else.

A receive also indicates both its unique ID and the unique ID of its nearest-enclosing wait (i.e., the first wait that witnesses the receive being completed). A suspended wait on a process indicates that the process is currently blocked at the the wait. The difference of the two wait commands is only meaningful in the abstract machine in Figure 5.4. Intuitively, the wait witnesses the completion of a receive on a process by marking it as matched. The wait is suspended if the next receive on the process for matching identifies a potential deadlock and can not be completed by the machine semantics.

$$
\begin{aligned}
ctp &::= (p\ \ldots) \\
p &::= (e\ \ldots\ \bot) \\
e &::= (\mathbf{s}\ x\ v\ v) \\
&\quad \mid (\mathbf{r}\ x\ y\ v\ x) \\
&\quad \mid (\mathbf{w}\ x) \\
&\quad \mid (*\mathbf{w}\ x) \\
&\quad \mid (\mathbf{b}\ x) \\
&\quad \mid (*\mathbf{b}\ x) \\
y &::= v \\
&\quad \mid * \\
v &::= \mathbf{number} \\
pt &::= (x\ \ldots) \\
x &::= \mathbf{ID}
\end{aligned}
$$

Figure 5.1: The language syntax for the abstract machine in Figure 5.4 – bold face indicates a terminal.

A barrier **b** is also associated with an identifier that is unique for its communicator. The communicator identifies a group of barrier commands and can only be used one time. A counted barrier is produced when a process arrives at the barrier. Processes block at the counted barrier until all members of the group have reached the barrier.

A pattern instance (pt) is a list of IDs indicating unique receive commands. Each process has at most one receive ID in the list. If a process has a receive ID in the list, then the receive attached to the ID marks a point of execution in the owning process. The intuition is that if each process is able to arrive at the indicated point of execution in the pattern instance, then a deadlock is imminent. If such a point of execution is arrived, the wait that witnesses the receive is suspended, indicating that all the commands preceding the receive may be scheduled to have the deadlock. All the commands following the receive should not be in the schedule. The suspended wait is not used anywhere other than identifying commands preceding a point of execution.

p_0	p_1	p_2
00 (**s** s_0 *0 1*)	10 (**r** r_1 $*$ *1* w_1)	20 (**r** r_3 *1 2* w_3)
01 (**r** r_0 $*$ *0* w_0)	11 (**w** w_1)	21 (**w** w_3)
02 (**w** w_0)	12 (**r** r_2 $*$ *1* w_2)	22 (**s** s_3 *2 0*)
03 (**s** s_1 *0 1*)	13 (**w** w_2)	
	14 (**s** s_2 *1 2*)	

Figure 5.2: A deadlock caused by circular dependency in messages.

5.2.2 Translation from MPI

Translating a program from MPI to a CTP requires the user to define the number of processes to run and a path condition for each process. The path condition can be determined either by instrumented runtime execution or symbolic evaluation.

The individual MPI procedures can then be translated from the operations in the program trace. A blocking send (`MPI_Send`) or recieve (`MPI_Recv`) becomes a send or recieve followed immediately by a wait. A non-blocking send (`MPI_Isend`) or recieve (`MPI_Irecv`) must be witnessed by a wait or test [3], which translates to a send or receive with the witness modeled by a wait. Barriers are translated directly.

5.2.3 Example

Figure 5.2 is an example CTP with line numbers and underlined commands to indicate a pattern instance. Process p_0 sends a message to p_1, then receives a message from any source, and finally sends another message to p_1. Process p_1 receives two messages from any source in the receives r_1 and r_2, and then sends a message to p_2. Process p_2 receives a message from p_1 and then sends a message to p_0. The tuple (r_0, r_2, r_3) is a pattern instance. If each process is able to arrive at these program points in some feasible execution, then the program contains the deadlock indicated by the pattern instance, which in this example, is a circular dependency.

Algorithm 2 Main Framework

```
1: PT ← PatternMatch(ctp, M)
2: for pt ∈ PT do
3:     (N_s, N_r, pt') ←FeasibleCheck(pt, ctp)
4:     if pt' ≠ ∅ then
5:         continue.
6:     else if ¬Validate(N_s, N_r, pt) then
7:         continue.
8:     else report error and exit.
9:     end if
10: end for
```

5.3 Main Framework

Algorithm 2 describes the general structure of the approach in this presentation which consists of three distinct steps: pattern matching (line 1), feasibility checking (line 3), and validating (line 6). PATTERNMATCH statically generates a set (PT) of matched pattern instances in the *ctp* with the help of an additional input, M, that defines all the potential match-pairs in the program. M can be generated in quadratic time [28].

FEASIBLECHECK is an abstract machine to prune pattern instances for which it is possible to prove that no feasible schedule exists. In other words, it is provably not possible to execute the *ctp* such that each process associated with a receive ID in the pattern instance is at that receive. The machine removes matched receive IDs from the pattern instance as it executes the *ctp*, so if pt' is not empty upon return, the pattern instance is provably not feasible and the algorithm continues with the next pattern instance (line 5). The algorithm additionally returns the number of issued sends (N_s) and the number of issued receives (N_r). These are used in validation depending on the type of the pattern instance: circular dependency or orphaned receive.

VALIDATE proves a pattern instance feasible, which means that it is a real deadlock in the *ctp*. If the deadlock is real, the algorithm reports the error and exits (line 8); otherwise it continues with the next pattern instance.

$$
\begin{aligned}
st &::= \texttt{(}ctp\ N_s\ N_r\ P_r\ N_b\ pt\texttt{)} \\
N_s &::= \emptyset \mid \texttt{(}N_s\ \texttt{[(}v\ v\texttt{)} \rightarrow v\texttt{]}\texttt{)} \\
N_r &::= \emptyset \mid \texttt{(}N_r\ \texttt{[(}v\ v\texttt{)} \rightarrow v\texttt{]}\texttt{)} \\
P_r &::= \emptyset \mid \texttt{(}P_r\ \texttt{[}x \rightarrow rcv\texttt{]}\texttt{)} \\
rcv &::= \texttt{([}x\ v\ v\texttt{]}\ \ldots\texttt{)} \\
N_b &::= \emptyset \mid \texttt{(}N_b\ \texttt{[}x \rightarrow v\texttt{]}\texttt{)}
\end{aligned}
$$

Figure 5.3: The machine syntax for the abstract machine in Figure 5.4.

FEASIBLECHECK is able to efficiently prune schedules that are provably non-feasible with predictive analysis using counting and auxiliary data structures to track FIFO ordering on messages.

Figure 5.4 is a term rewriting system for a syntactic machine (i.e., the machine state is represented by a string) for FEASIBLECHECK to prove that a pattern instance is not feasible. Figure 5.3 is the syntax for that machine. The rewrites define how the machine executes an input *ctp* and pattern instance *pt* by evolving the machine state. At a high-level, the machine can

- Process a send, receive, or barrier from a process by counting the send (*Sndi Command*), queuing up the receive on the indicated wait ID (*Rcvi Command*), or counting the barrier (*Barrier Command 1*).
- Consume a wait from a process when the associated queue is empty (*Wait (Rcvi) Command 1*)
- Remove a receive ID from the pattern instance and suspend the indicated wait when the associated receive is next on the queue for the wait (*Wait (Rcvi) Command 2*).
- Remove the next receive from the queue for the indicated wait and update the number of receives when feasible (*Wait (Rcvi) Command 3*).
- Consume a barrier if all the processes have arrived (*Barrier Command 2*) – communicator groups may only be used once.

SNDI COMMAND

$$\frac{N_s(v_{to}, v_{frm}) = v_c \quad N_s(v_{to}, *) = v_i \qquad {N_s}' = N_s[(v_{to}, v_{frm}) \mapsto v_c + 1,\ (v_{to}, *) \mapsto v_i + 1]}{((p_0\ \ldots\ ((\mathbf{s}\ x\ v_{frm}\ v_{to})\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ (e_1\ \ldots\ \bot)\ p_2\ \ldots)\ {N_s}'\ N_r\ P_r\ N_b\ pt)}$$

RCVI COMMAND

$$\frac{P_r(x_w) = ([x_1\ v_{frm1}\ v_{to1}]\ \ldots) \qquad {P_r}' = P_r[x_w \mapsto ([x_1\ v_{frm1}\ v_{to1}]\ \ldots\ [x_0\ v_{frm0}\ v_{to0}])]}{((p_0\ \ldots\ ((\mathbf{r}\ x_0\ v_{frm0}\ v_{to0}\ x_w)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ (e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ {P_r}'\ N_b\ pt)}$$

WAIT (RCVI) COMMAND 1

$$\frac{P_r(x_w) = ()}{((p_0\ \ldots\ ((\mathbf{w}\ x_w)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ (e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt)}$$

WAIT (RCVI) COMMAND 2

$$\frac{P_r(x_w) = ([x_0\ v_{frm0}\ v_{to0}]\ [x_1\ v_{frm1}\ v_{to1}]\ \ldots) \quad pt = (x_a\ \ldots\ x_0\ x_b\ \ldots) \qquad x_0 \in pt \qquad pt' = (x_a\ \ldots\ x_b\ \ldots)}{((p_0\ \ldots\ ((\mathbf{w}\ x_w)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ ((*\mathbf{w}\ x_w)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt')}$$

WAIT (RCVI) COMMAND 3

$$\frac{\begin{array}{l} P_r(x_w) = ([x_0\ v_{frm0}\ v_{to0}]\ [x_1\ v_{frm1}\ v_{to1}]\ \ldots) \qquad x_0 \notin pt \\ N_r(v_{to0}, v_{frm0}) < N_s(v_{to0}, v_{frm0}) \qquad N_r(v_{to0}, *) < N_s(v_{to0}, *) \qquad N_r(v_{to0}, v_{frm0}) = v_c \\ {N_r}' = N_r[(v_{to0}, v_{frm0}) \mapsto v_c + 1] \qquad {P_r}' = P_r[x_w \mapsto ([x_1\ v_{frm1}\ v_{to1}]\ \ldots)] \end{array}}{((p_0\ \ldots\ ((\mathbf{w}\ x_w)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ ((\mathbf{w}\ x_w)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ {N_r}'\ {P_r}'\ N_b\ pt)}$$

BARRIER COMMAND 1

$$\frac{N_b(x_0) = v_c \qquad v_c < N_{proc} \qquad {N_b}' = N_b[x_0 \mapsto v_c + 1]}{((p_0\ \ldots\ ((\mathbf{b}\ x_0)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ ((*\mathbf{b}\ x_0)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ {N_b}'\ pt)}$$

BARRIER COMMAND 2

$$\frac{N_b(x_0) = N_{proc}}{((p_0\ \ldots\ ((*\mathbf{b}\ x_0)\ e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt) \rightarrow_m ((p_0\ \ldots\ (e_1\ \ldots\ \bot)\ p_2\ \ldots)\ N_s\ N_r\ P_r\ N_b\ pt)}$$

Figure 5.4: Machine reductions ($\rightarrow_m$).

At a lower-level, the machine state (st) is a six-tuple of variables. The first variable ctp defines the concurrent trace program being analyzed. The set N_s maps a destination endpoint and a source endpoint to a number that is used to count issued sends. The variable N_r has the same structure only the number is used to count the number of matched receives. The variable P_r records the pending receives by mapping the unique identifier of a wait to a queue of the issued receives rcv. In this queue, the action identifier, the source endpoint and the destination endpoint are recorded for each receive. The variable N_b maps the unique identifier of a communicator to a number that is used to count the number of witnessed barriers.

The *Sndi Command* in Figure 5.4 consumes sends in a process. $N_s{}'$ is a new set, just like the old set N_s, only the new set maps the destination endpoint v_{to} and the source endpoint v_{frm} to the number v_c+1 where v_c is the value in the old set on the same (v_{to}, v_{frm}) key. It also maps v_{to} and $*$ to the number v_i+1 where v_i is the value in the old set on the same $(v_{to}, *)$ key. As mentioned previously, the notation $*$ is a special source endpoint indicating any source.

The *Rcvi Command* in Figure 5.4 consumes receives by updating P_r. Similar to the rule *Sndi Command*, P_r merely inserts a new record for the receive x_0 to the tail of the queue indexed by the wait that witnesses the completion of x_0.

The *Wait (Rcvi) Command* operates in three ways. If the wait x_w maps to an empty queue in P_r, indicating that no receives need to be completed by x_w, then x_w is simply consumed (*Wait (Rcvi) Command 1*).

If the next receive x_0 in the queue $P_r(x_w)$ is found in pt where the notation $\in$ is used to indicate this condition, then x_0 is removed from pt and the wait x_w is suspended in a process by replacing the notation $\mathbf{w}$ with $*\mathbf{w}$, meaning that x_0 in pt is reached and this process blocks at x_w in execution (*Wait (Rcvi) Command 2*).

The last rule for wait (*Wait (Rcvi) Command 3*) checks whether the next receive x_0 in the queue $P_r(x_w)$ is able to be consumed. The rule is only active if x_0 is not in pt. The rule requires two conditions:

$$N_r(v_{to0}, v_{frm0}) < N_s(v_{to0}, v_{frm0})$$

checks whether there are more counted sends than counted receives with common source and destination endpoints, and

$$N_r(v_{to0}, *) < N_s(v_{to0}, *)$$

checks whether there are more counted sends than counted receives for any preceding wild-card receives. These conditions indicate that at least one send can be matched with x_0. If

$$((((\mathrm{s}\ s_0\ 0\ 1)\ (\mathrm{r}\ r_0\ *\ 0\ w_0)\ \ldots\ \bot)\ p_1\ p_2)\ (((mt\ [(1\ 0) \rightarrow 0])\ [(1\ *) \rightarrow$$
$$0])\ \ldots)\ N_r\ P_r\ N_b\ (\mathrm{r}_0\ \mathrm{r}_2\ \mathrm{r}_3)) \rightarrow_m ((((\mathrm{r}\ r_0\ *\ 0\ w_0)\ \ldots\ \bot)\ p_1\ p_2)\ (((mt\ [(1\ 0) \rightarrow 1])\ [(1\ *) \rightarrow$$
$$1])\ \ldots)\ N_r\ P_r\ N_b\ (\mathrm{r}_0\ \mathrm{r}_2\ \mathrm{r}_3))$$

Figure 5.5: An example of machine reduction.

both conditions hold, then N_r is updated with the counted receive, and x_0 is removed from P_r.

The *Barrier Command* moves the barrier forward by its synchronization rule. It is assumed that the group of any barrier consists of all the processes in the *ctp*. If the count of the witnessed barriers $N_b(x_0)$ for a specific communicator x_0 is less than the total number of the processes (N_{proc}), indicating that the barriers for x_0 are not matched, then the barrier is not consumed. Rather, the barrier is counted by incrementing $N_b(x_0)$ and replacing the notation **b** with $*$**b** for the same communicator in the *ctp* (*Barrier Command 1*).

If $N_b(x_0)$ is equal to N_{proc} indicating that all the barriers for x_0 are matched, then they can be consumed (*Barrier Command 2*).

The machine rewrites the state until no more reduction rules can be applied indicating that there is no way to further execute the program. The first statement that cannot be consumed on any process is either the bottom of the process or a blocking command. A blocking command could be a wait or a barrier. If at the end, there are receive IDs in the pattern instance, then the pattern instance is provably non-feasible. In such a case, the machine *accepts* the program as free of deadlock on the pattern instance; otherwise, the machine *rejects* the program as having a deadlock on the pattern instance.

Figure 5.5 is an example of the machine reduction given the CTP in Figure 5.2. The reduction applies the *Sndi Command* in Figure 5.4 to consume the send s_0 on process p_0. The reduction shows that s_0 is removed from the *ctp* in the state. Also, the count of issued sends is incremented by one on the keys (1 0) and (1 $*$), respectively.

The soundness for the abstract machine is given in Lemma 4.

Lemma 4. *The machine implmenting* FEASIBLECHECK *is sound in that it only accepts programs that do not deadlock on the associated pattern instance (but it may reject some programs as having a deadlock on that instance when in fact they do not).*

Proof. The commands defined in Figure 5.5 must never claim to be able to consume a send, a receive, or a barrier incorrectly for the machine to be sound.

Some rules are trivial to prove. In precise, *Send Command* and *Receive Command* simply consume a non-blocking send or a non-blocking receive according to the semantics. The rules also update the associated counts N_s and P_r with correct values. *Wait (RCVI) Command 1* consumes a wait because there is no receives in P_r to be witnessed by the wait. *Wait (RCVI) Command 2* suspends a wait because the next receive that the wait witnesses is in the pattern instance. The program that reaches all the receives in the pattern instance may have a deadlock and is rejected by the machine. *Barrier Command 1* suspends a barrier because not all the barriers in the same group are witnessed. *Barrier Command 2* consume a suspended barrier because all the barriers in the same group are witnessed.

Only *Wait (RCVI) Command 3* is non-trivial to prove. The rule uses two conditions $N_r(v_{to0}, v_{frm0}) < N_s(v_{to0}, v_{frm0})$ and $N_r(v_{to0}, *) < N_s(v_{to0}, *)$ to check if the receive x_0 can be consumed. Note that the messages are received in a FIFO order on any process. If x_0 is a wildcard receive on the process v_{to0}, the two conditions are equivalent, both checking if there exists at least one send that can match x_0 after all the preceding wildcard receives on v_{to0} are matched. If x_0 is a deterministic receive on the process v_{to0}, then the first condition checks if there is at least one send that can match x_0 and all the preceding deterministic receives with identical source and destination are matched. The second condition checks if all the preceding wildcard receives on v_{to0} are matched so that the message FIFO order is preserved. Both cases imply that x_0 can be consumed correctly. Therefore, the soundness of the machine is proved. □

Corollary 1. *The machine implementing the function* FEASIBLECHECK *never miss counts the number of sends, receives matched and barriers: N_r, N_s and N_b are correct.*

p_0	p_1	p_2
00 (**s** s_0 *0 1*)	10 (**r** r_1 * *1* w_1)	20 (**r** r_2 *1 2* w_2)
01 (**r** r_0 * *0* w_0)	11 (**w** w_1)	21 (**w** w_2)
02 (**w** w_0)	12 (**s** s_2 *1 2*)	22 (**s** s_3 *2 0*)
03 (**s** s_1 *0 1*)		

Figure 5.6: No deadlock caused by circular dependency in messages.

Proof. The *Sndi Command* increments count every time the rule activates and updates both the counter for the specified endpoints and special counter that records sends that can match with wildcard receives. The *Wait (Rcvi) Command 3* only increments the indicated receive counter when it is possible to consume the receive and by Lemma 4, the machine only matches in a way that is sound. The *Barrier Command 1* only increments the indicated barrier counter when the barrier is witnessed. The counters are not incremented in any other rule. Therefore, nothing is missed, and nothing is double counted. □

Corollary 2. *Algorithm 2 is sound and complete if and only if the function* PATTERNMATCH *is complete (i.e., it gives all pattern instances and possibly more) and the function* VALIDATE *is sound and complete (i.e., any deadlock it detects is a real one and no real deadlocks are rejected).*

The presentation further discusses the completeness of the function PATTERNMATCH and the soundness and completeness of the function VALIDATE for two distinct deadlock patterns, circular dependency and orphaned receive.

5.4 Circular Dependency

The circular dependency represents a cycle among a group of processes in a CTP. A deadlock occurs when a receive on each process in the cycle waits for the issuing of a send on another member process but never gets a response.

The circular dependency in Definition 30 is from the definition of lock dependency [30]. It depends on the sequential relation in Definition 29.

Definition 29. *A sequential relation for a process, p, is a three-tuple (p, r_c, s_l), where the receive r_c and its nearest-enclosing wait are both followed by the send s_l on p.*

Definition 30. *Given a set of sequential relations, D, a circular dependency $\tau = \langle (p_0, r_0, s_0), \ldots, (p_m, r_m, s_m) \rangle$ is a sequence in D, such that the following properties hold.*

1. *at least two sequential relations exist in τ;*
2. *for all integers $i, j \in [0, m]$, $p_i \neq p_j$, i.e., the processes p_0, p_1, ..., p_m are all distinct objects;*
3. *for all $i \in [0, m]$, $j = (i + 1)\%m$, the send s_i can potentially match the receive r_j;*

In Figure 5.2, the CTP has a circular dependency in messages $\langle (p_0, r_0, s_1), (p_1, r_2, s_2), (p_2, r_3, s_3) \rangle$ that may cause the CTP to deadlock. For instance, r_0 can never match s_3 because of the dependency. For simplicity, an instance of the circular dependency pattern only records the receives. For example, (r_0, r_2, r_3) is an instance of the circular dependency pattern for the CTP in Figure 5.2. As a note, this section considers only infinite buffer semantics in the discussion of algorithms and examples. The zero buffer semantics are discussed in section 6.

5.4.1 Pattern Match for Circular Dependency

Algorithm 3 shows the steps for finding all the instances of the circular dependency pattern in a *ctp*. As a reminder, M is the set of potential match pairs for the *ctp*. In general, the algorithm is part of the function PATTERNMATCH in Algorithm 2. It first maps a *ctp* to a graph that consists of a set of vertices V and a set of edges E. It then detects all the cycles in the graph based on Johnson's algorithm [29].

The set PT stores the matched pattern instances. The set R stores the witnessed receives on a process that is represented as a list of operations $(e_1, e_2, \ldots, e_n)$. The set R_w stores the receives that are recently witnessed on p.

Algorithm 3 Finding Circular Dependency

```
1: PT ← ∅
2: E ← ∅, V ← ∅
3: R ← ∅, R_w ← ∅
4: for (e_1, e_2, ..., e_n) ∈ ctp do
5:     for i ← 1 to n do
6:         if e_i is a receive then
7:             R = R ∪ {e_i}
8:             V = V ∪ {e_i}
9:             for s_l ∈ M(e_i) do                    ▷ add match relation
10:                E = E ∪ {(s_l, e_i)}
11:            end for
12:        end if
13:        if e_i is a wait then
14:            R_w = R_w ∪ R
15:            R ← ∅
16:        end if
17:        if e_i is a send then
18:            V = V ∪ {e_i}
19:            for r_c ∈ R_w do                       ▷ add HB relation on r_c to e_i
20:                E = E ∪ {(r_c, e_i)}
21:            end for
22:        end if
23:    end for
24:    R ← ∅, R_w ← ∅
25: end for
26: PT ← JOHNSON(V, E)
```

The function $M(r_c) = \{s_l \mid \langle r_c, s_l \rangle \in M\}$ returns all potential sends for the receive. The algorithm checks each operation e_i in p. If e_i is a receive, then it is inserted into R at line 7 and is inserted into V at line 8. Also, for each potentially matched send s_l in $M(e_i)$, the relation (s_l, e_i) is inserted into E at line 10 indicating that a match relation on s_l to e_i is an edge in the graph. If e_i is a wait, all the receives in R are inserted to R_w at line 14 and R is set back to an empty set at line 15, indicating that all the receives in R are witnessed by e_i. Definition 29 requires for each sequential relation that the receive and its nearest-enclosing wait both precede the send on an identical process. Therefore, the happens-before relation (a partial order over two operations) on a witnessed receive to a following send on an identical process is considered as an edge in the graph because they can be potentially matched as a sequential relation. As such, if e_i is a send, the algorithm checks each witnessed receive $r_c \in R_w$ at line 19, and inserts the happens-before relation (r_c, e_i) into E at line 20.

Finally, the function JOHNSON implements Johnson's algorithm to compute all the cycles in the graph. Since an edge in any computed cycle is either a receive-send happens-before relation on an identical process, or a send-receive match relation across two processes, there exists exactly one sequential relation for any process in the cycle. Therefore, each cycle computed by the function JOHNSON is an instance of the circular dependency pattern. As discussed earlier, only receives in each cycle are stored to represent an instance in the set PT. The complexity of program traversal is $O(\mathrm{N}^2)$, where N is the total number of operations in the program. The complexity of Johnson's algorithm is $O((|V| + |E|)(\mathrm{c} + 1))$ $\approx O((\mathrm{c} + 1)\mathrm{N}^2)$, where c is the number of cycles. Therefore, the total complexity of the algorithm is $O((\mathrm{c} + 1)\mathrm{N}^2)$.

Figure 5.7 shows the graph that is built on the example CTP in Figure 5.2 by Algorithm 3. In the graph, the vertices include s_0, r_0, s_1, r_1, r_2, s_2, r_3, and s_3. The edges include the match relations (s_3, r_0), (s_0, r_1), (s_1, r_2), and (s_2, r_3), and the happens-before relations (r_0, s_1), (r_1, s_2), (r_2, s_2), and (r_3, s_3). As shown in the graph, a cycle

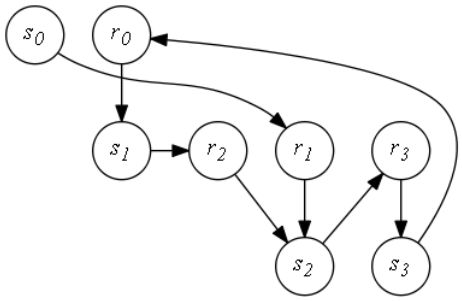

Figure 5.7: The graph built on the CTP in Figure 5.2 by Algorithm 3.

$r_0 \rightarrow s_1 \rightarrow r_2 \rightarrow s_2 \rightarrow r_3 \rightarrow s_3 \rightarrow r_0$ exists where the receives are stored as an instance of the circular dependency pattern.

Lemma 5. *The pattern match in Algorithm 3 is complete indicating that all possible pattern instances are detected (it may include some instances that do not have deadlocks).*

Proof. The graph built by Algorithm 3 consists of all possible vertices and all possible edges because the algorithm statically traverses each process from the beginning to the end and adds an edge once the relation is detected. Also, Johnson's algorithm is able to compute all the cycles in the graph. Therefore, all possible instances for the circular dependency pattern are detected. □

5.4.2 Validation for Circular Dependency

If the function FEASIBLECHECK demonstrates that a potentially feasible schedule exists, the deadlock is further validated by (5.1) for the pattern instance *pt*.

$$\forall r_c \in pt(N_s(dest, src) \leq N_r(dest, src)) \tag{5.1}$$

where *dest* is the destination endpoint of the receive r_c, and *src* is the source endpoint of r_c. The formula checks whether there exists a send that may match any receive r_c in *pt* by comparing the count of issued sends and the count of issued receives. The function $N_s(dest, src)$ returns the count of issued sends with the destination *dest* and the source *src*.

The function $N_r(dest, src)$ returns the count of issued receives with the destination *dest* and the source *src*. If the formula is satisfied, then the algorithm detects a real deadlock for *pt*. If the formula is unsatisfiable for any receive r_c in *pt*, then no real deadlock exists for *pt*.

The CTP in Figure 5.6 is an example that has no deadlock for the circular dependency pattern instance. In Figure 5.6, process p_0 sends a message to p_1, receives a message from any source, and then sends another message to p_1. Process p_1 receives a message from any source, and then sends a message to p_2. Process p_2 receives a message from p_1, and then sends a message to p_0. As shown, the underlined commands represent an instance of the circular dependency (r_0, r_1, r_2). However, (5.1) is not satisfied because the send s_0 matches the receive r_1 and the cycle does not exist any more.

Lemma 6. *The validation in (5.1) for an instance of the circular dependency pattern, pt, is sound and complete indicating that any detected deadlock is a real deadlock and any instance that does not satisfy (5.1) is a not a real deadlock.*

Proof. Since the counts stored in N_s and N_r are correct (Corollary 5.3), (5.1) is able to correctly check if an instance is a real deadlock or not. If the formula is unsatisfiable for some receive r_c, then r_c can be matched and the dependency in the cycle does not exist. Therefore, *pt* does not imply a real deadlock. If the formula is satisfied, then no receive in *pt* can be matched. Therefore, a real deadlock occurs for *pt*. □

5.5 Orphaned Receive

A deadlock may occur when the runtime matches a wildcard receive with the send needed by a deterministic receive.

Figure 5.8 is an example CTP that deadlocks for an orphaned receive pattern instance. The underlined command indicates a pattern instance. Process p_0 sends a message to p_1, and then receives two messages from any source and p_1 in the receive r_0 and r_1. Process p_1 receives a message from any source, and then sends a message to p_0. Process p_2 sends a

p_0	p_1	p_2
00 (**s** s_0 *0 1*)	10 (**r** r_2 $*$ *1* w_2)	20 (**s** s_2 *2 0*)
01 (**r** r_0 $*$ *0* w_0)	11 (**w** w_2)	
02 (**w** w_0)	12 (**s** s_1 *1 0*)	
03 (**r** r_1 *1 0* w_1)		
04 (**w** w_1)		

Figure 5.8: A deadlock caused by orphaned receive.

message to p_0. A deadlock may occur if the receive r_0 is matched with the send s_1 making it unable to match the receive r_1 with the send s_2. This CTP shows an instance of the orphaned receive pattern in Definition 31.

Definition 31. *An orphaned receive is a pair* (r_w, r_c) *where,*

1. r_w *is a wildcard receive and* r_c *is a deterministic receive that follows* r_w *on an identical process p;*
2. *at least two sends from different source endpoints other than p may potentially match* r_w*; and*
3. *among these matched sends, at least one send matches* r_c*.*

For simplicity, only the deterministic receive is stored to represent an instance of the orphaned receive pattern. For example, the CTP in Figure 5.8 has an orphaned receive pattern instance (r_1). As a note, the presentation in this section is only for infinite buffer semantics. The zero buffer semantics are discussed in section 6.

5.5.1 Pattern Match for Orphaned Receive

Algorithm 4 shows the steps of finding the orphaned receive pattern instances for an input *ctp*. The algorithm is part of the function PATTERNMATCH in Algorithm 2. It checks all the pairs of wildcard/deterministic receives on each process, and then finds all the instances that satisfy the criteria in Definition 31.

Algorithm 4 Finding Orphaned Receive

```
1: PT ← ∅
2: R ← ∅
3: for (e_1, e_2, ..., e_n) ∈ ctp do
4:     for i ← 1 to n do
5:         if e_i is a wildcard receive then
6:             R = R ∪ {e_i}
7:         end if
8:         if e_i is a deterministic receive then
9:             for r_w ∈ R do
10:                if M(e_i) ∩ M(r_w) ≠ ∅ ∧ |M(r_w)| > 1 then
11:                    PT = PT ∪ {{e_i}}
12:                end if
13:            end for
14:        end if
15:    end for
16:    R ← ∅
17: end for
```

The set PT stores the matched pattern instances. The set R stores the witnessed wildcard receives on the process p that is a list of operations $(e_1, e_2, \ldots, e_n)$.

The algorithm checks each operation e_i in p. If e_i is a wildcard receive, then it is inserted into R indicating that it is witnessed. Definition 31 requires that the wildcard receive in an orphaned receive pair happens before the deterministic receive in the same pair on an identical process. Therefore, only a wildcard receive with a following deterministic receive is able to build an orphaned receive pattern instance. If e_i is a deterministic receive, it checks each wildcard receive r_w in R. If the condition at line 10 is also satisfied such that r_w and e_i have a common set of potential sends and more than one send can be matched with r_w, a new instance $\{e_i\}$ is added to PT at line 11. R is set back to empty at line 16, indicating that the algorithm starts at a new process for the pairs of wildcard/deterministic receives. The complexity of the algorithm is $O(\mathrm{N}^2)$, where N is the total number of operations in the *ctp*.

Lemma 7. *The pattern match in Algorithm 4 is complete indicating that all possible pattern instances are detected (it may include some instances that do not have deadlocks).*

Proof. Algorithm 4 considers all possible pairs of wildcard/deterministic receives on each process. Also, the deterministic receive cannot be orphaned for any pair that does not satisfy the condition at line 10. This is because 1) there does not exist any common send that can potentially match the wildcard receive and the deterministic receive, or 2) there is no non-deterministic choice for the wildcard receive to be matched with. Therefore, all possible instances of the orphaned receive pattern are detected. □

5.5.2 Validation for Orphaned Receive

Algorithm 5 Validate Orphaned Receive

```
1: if SAT(ENCODE(ctp_s, pt, M)) then
2:     report deadlock and exit.
3: end if
```

The validation in Algorithm 5 is more complicated than the validation for the circular dependency because it requires a higher cost SMT encoding for the ctp_s that is generated by a modification of the abstract machine in Figure 5.4. This modification executes a CTP just like the old machine, only it updates a new state member ctp_s, by adding the consumed operations when executing the machine. The function ENCODE encodes the ctp_s into an SMT problem based on the rules in [28]. Further, a new rule is added to the encoding: for the receive r_c in the pattern instance pt, $\bigvee_{r_i \in M(s_l)} \langle r_i, s_l \rangle$ is encoded for any send $s_l \in M(r_c)$ in ctp_s. This rule ensures that no send in ctp_s can match r_c. The function SAT is true if the encoding is satisfiable. The existence of a satisfying assignment of the encoding implies a real deadlock for pt. If the encoding is unsatisfiable, no deadlock exists for pt.

The CTP in Figure 5.9 is an example that has no deadlock for the orphaned receive pattern instance. In Figure 5.9, process p_0 receives a message from any source, sends a message to p_1, and then receives a message from p_1. Process p_1 receives a message from any source, and then sends a message to p_0. Process p_2 sends a message to p_0. As shown, the pair (r_0, r_1) satisfies the criteria in Definition 31, therefore, the instance $\{r_1\}$ is matched by Algorithm 4. However, the encoding generated by the function ENCODE in Algorithm 5 is

p_0	p_1	p_2
00 (**r** r_0 $*$ *0* w_0)	10 (**r** r_2 $*$ *1* w_2)	20 (**s** s_2 *2* *0*)
01 (**w** w_0)	11 (**w** w_2)	
02 (**s** s_0 *0* *1*)	12 (**s** s_1 *1* *0*)	
03 (**r** r_1 *1* *0* w_1)		
04 (**w** w_1)		

Figure 5.9: No deadlock caused by orphaned receive.

unsatisfiable for the instance $\{r_1\}$ because the receive r_0 can not match the send s_1 so the receive r_1 is matched eventually.

Lemma 8. *The validation method implementing Algorithm 5 for an instance of the orphaned receive pattern, pt, is sound and complete indicating that any detected deadlock is a real deadlock and any instance rejected by Algorithm 5 is not a real deadlock.*

Proof. The soundness and completeness relies on the correctness proof of the SMT encoding [28]. In general, a satisfying assignment returned by Algorithm 5 represents a feasible schedule leading to an orphaned receive. Therefore, a real deadlock is detected. If the SMT encoding is unsatisfiable indicating that no schedules can be resolved for *pt*, then no real deadlock may occur. □

5.6 Zero Buffer Semantics

The circular dependency and the orphaned receive for infinite buffer semantics are discussed in section 4 and 5, respectively. The two patterns may also cause a program deadlock under zero buffer semantics. Notice that the rules in Figure 5.4 are consistent with how messages communicate under infinite buffer semantics (e.g., a send is issued immediately). The zero buffer semantics, however, use a different way of message communication. Therefore, a schedule validated by the function VALIDATE in Algorithm 2 may not be feasible for zero buffer semantics.

To check if a program may deadlock for the circular dependency or the orphaned receive under zero buffer semantics, the zero buffer compatibility [25] should be checked with two modifications to the validation. First, the function ENCODE in Algorithm 5 is extended to support zero buffer semantics based on the encoding rules in [25]. This change ensures that a satisfying assignment of the encoding represents a schedule that is compatible with zero buffer semantics for an orphaned receive pattern instance. Second, the validation for the circular dependency is modified to use the same SMT encoding to check the feasibility of a CTP generated by the abstract machine. This is because a deadlock for a circular dependency pattern instance under zero buffer semantics has a zero buffer compatible schedule.

5.7 Experiments

The experiments compare the performance of the approach in this presentation with two state-of-art MPI verifiers MOPPER [20], a SAT based tool, and ISP [49, 58], a dynamic analyzer.

MOPPER is designed to verify only single-path programs. ISP can be applied to both single-path programs and programs with branching. Therefore, it is meaningful only when the benchmarks are single-path. The initial CTP for the approach in this paper is generated by instrumenting the MPI programs with manually written scripts and executing the programs by MPICH [39], a public implementation of the MPI standard. The SMT encoding for the approach in this presentation, if needed, is resolved by the SMT solver Z3 [13]. Similarly, MOPPER launches ISP to automatically generate the initial program trace as input.

A series of experiments are conducted for a set of single-path programs, including three small programs [20] that each contains a deadlock, and seven typical benchmark programs employed by other papers [1, 8, 65]. All the results show the comparison under infinite buffer semantics. The experiments are run on a AMD A8 Quad Core processor with 6 GB

Table 5.1: Tests on Selected Benchmarks

Test Programs					Our Method				ISP		MOPPER	
Name	#Procs	#Calls	Match	Deadlock	PT	PTR	D	Time	D	Time	D	Time
dlg1	3	8	2	Yes	1	0	√	0.009s	√	0.116s	√	0.001s
dlg5	3	16	12	Yes	1	0	√	0.021s	√	0.118s	√	0.002s
dlg8	3	12	4	Yes	1	0	√	0.019s	√	0.110s	√	0.002s
Monte	4	35	24	No	0	0		0.001s		0.957s		0.015s
	8	75	40K k	No	0	0		0.002s		TO		0.751s
	16	155	2E13	No	0	0		0.006s		TO		TO
Integrate	8	36	5K	No	0	0		0.001s		>1000s		0.103s
	10	46	362K	No	0	0		0.002s		TO		34.986s
	16	76	1E12	No	0	0		0.003s		TO		TO
Diffusion2D	4	52	6E9	No	0	0		0.003s		32.005s		0.039s
	8	108	2E21	No	0	0		0.004s		TO		TO
Floyd	8	120	4E29	No	0	0		0.004s		TO		2.812s
	16	256	1E58	No	0	0		0.006s		TO		62.467s
GE	8	56	64	No	0	0		0.011s		1.054s		0.042s
	16	120	16K	No	0	0		0.014s		1.426s		0.098s
Mismatch	3	400	100	Yes	50	49	√	1.609s	√	4.274s	√	2.601s
	3	800	2E40	Yes	100	98	√	11.027s	√	514.852s	√	17.892s
Circular	3	252	8E257	Yes	132,651	130,804	√	13.821s		TO	√	728.722s

of memory running Ubuntu 14.04 LTS. A time limit of 30 minutes is set for each test. The test aborts the verification process if it does not complete within the time limit.

The results of the comparison are in Table 5.1. The column "Match" records the approximated number of match possibilities. A program with a large number of match possibilities has a large degree of message non-determinism. The column "Deadlock" indicates the existence of deadlocks. The column "PT" is the number of pattern instances that the algorithm in this presentation detects. The column "PTR" is the number of pattern instances that are pruned by the approach in this presentation. The column "D" indicates whether the tool detects a deadlock or not. The "Time" column for the approach in this presentation is the time of static analysis and constraint solving (if necessary). The "Time" column of MOPPER is for constraint generation and solving. The column "Time" for ISP is the running time of dynamic analysis. The notation "TO" means "time out" (exceeding the time limit set for each test).

The experiments are launched for a set of benchmarks. The small tests include *dlg1, dlg5 and dlg8* that implement simple message communications [20]. Each contains a deadlock for the orphaned receive pattern.

The typical benchmarks are tested as well. *Monte* implements the Monte Carlo method to compute π [8]. It uses one manager process and multiple worker processes to send messages back and forth. In addition, barrier operations are used to synchronize the program.

Integrate uses heavy non-determinism in message communication to compute an integral of the sin function over the interval $[0, \pi]$ [1]. This benchmark also has a manager-worker pattern where the root process divides the interval to a certain number of tasks. It then distributes those tasks to multiple worker processes.

Diffusion2D has an interesting computation pattern that uses barriers to "partition" the message communication into several sections [1]. A message from a send can be only received in a common section.

Floyd implements the shortest path algorithm for all the pairs of nodes [65]. Each node communicates only with the immediate following neighbor.

GE is a message passing implementation for Gaussian Elimination [65]. Messages are communicated by issuing several wildcard receives on each node.

Mismatch implements the message communication that contains a set of the orphaned receive pattern.

Circular implements the message communication that contains a deadlock for the circular dependency pattern.

The results show that the algorithm in this presentation is more efficient than ISP and MOPPER. For the small programs (e.g., *dlg1*), all the tools correctly find the deadlocks and return very fast. For the large benchmark programs that do not have any deadlock (e.g., *Monte*), ISP runs much slower than MOPPER and the approach in this presentation. For instance, it runs out of time when the program size of *Monte* increases to 8 processes. MOPPER runs fast when the number of processes is small, however, it may time out with a large number of processes. The approach in this presentation, however, returns very fast even with a large number of processes. This is because it does not need to check feasibility

or validate any pattern instances once it detects no instances are matched. For the large benchmark programs with deadlocks, the approach in this presentation remains faster than other tools, even though the higher cost SMT encoding is used this time. Also, the approach in this paper is able to prune a large number of non-feasible pattern instances. For example, the program *Circular* prunes 130,804 non-feasible pattern instances out of 132,651 instances and detects a real deadlock. Since all the deadlocks in the tests are detected by the approach in this presentation, it is believed that the two deadlock patterns may cover many deadlock cases in MPI programs.

5.8 Related Work

The approach in this presentation is inspired by several works. The predictive analysis collects a single trace and predicts deadlocks in the other traces with the same input [46, 50]. The dependency constructor in the work refines the match pairs that may lead to a deadlock. The refining strategy uses simple counting rules that inspires the abstract machine in this presentation. However, the approach in this presentation does not refine the match pairs for deadlock detection.

Joshi et al. proposed a method that finds real deadlocks for multi-threaded Java programs by first detecting potential lock dependency cycles with a imprecise dynamic analyzer and then finding real deadlocks by a random thread scheduler with high probability [30]. The solution scales to large programs. Also, the method detects a number of previously unknown deadlocks in a set of benchmarks. The refining strategy of the work also inspires the general algorithm in this presentation, only this presentation intends to prune a set of potential deadlock instances instead of finding deadlocks from an instance.

A precise SMT encoding technique is proposed for detecting user-provided assertions for MCAPI programs [28]. The encoding is sound and complete and is easy to use to reason about infinite buffer semantics without requiring a precise match set. The work also provides an algorithm that runs in quadratic time complexity to generate a sufficiently small over-

approximated match set based on the given execution trace. This approach is extended to checking zero buffer incompatibility for MPI semantics [25]. The technique is also used for deadlock detection for MPI programs in this presentation.

There are other solutions for message passing program analysis. The dynamic analyzer ISP implements the POE algorithm, a Dynamic Partial Order Reduction (DPOR) algorithm [19] applied to MPI programs [58]. An extension is the MSPOE algorithm [49]. It operates by postponing the cooperative operations for message passing in transit until each process reaches a blocking call. It then determines the potential matches of send and receive operations in the runtime. The solution is able to detect errors such as assertion violation and deadlock in an MPI program. A drawback of ISP is that it does not scale for large programs due to state explosion.

Forejt et al. proposed a SAT based approach to detect deadlock in a single-path MPI program [20]. The solution is correct and efficient for programs with a low degree of message non-determinism. However, since the size of the encoding is cubic, checking large programs is time consuming. The SMT encoding used in this presentation, however, is quadratic. Similar to the solution in this presentation, the work requires a match pair set.

Umpire is an approach of runtime verification for checking multiple MPI errors such as deadlock and resource tracking [60]. The error checking is taken by spawning one manger thread and several outfielder threads in the execution of an MPI program. A drawback of the approach is that it relies on a concrete execution, which may miss the errors in the other execution trace. An extension to Umpire is Marmot [32]. The work uses a centralized sever instead of multiple threads for error checking. Another extension to Umpire is MUST [23]. The structure of MUST allows the users to execute the error checking either in an application process itself or in extra processes that are used to offload these analyses. However, just like Umpire and Marmot, the approach is neither sound nor complete for deadlock detection.

MPI-Spin is integrated in the model checker SPIN [24], for verifying MPI programs [51, 52]. It generates a model of an MPI program and symbolically executes it. It does not scale to large programs with a large degree of message non-determinism.

Vo et al. used Lamport clocks to update the auxiliary information via piggyback messages [62, 63]. While completeness is abandoned in their analysis, they show the work is useful and efficient in practice.

Sharma et al. proposed the first push button model checker for MCAPI – MCC [48]. It indirectly controls the MCAPI runtime to verify MCAPI programs under zero buffer semantics. An obvious drawback of the work is its inability to analyze infinite buffer semantics which is known as a common runtime environment in message passing. A key insight, though, is the direct use of match pairs – couplings for potential sends and receives.

Elwakil et al. also used SMT techniques to reason about the program behavior in the MCAPI domain [17, 18]. State-based and order-based encoding techniques are both used. These techniques fail to reason about the infinite buffer semantics and require a precise match set which is non-trivial to compute beforehand.

5.9 Conclusion and Future Work

This presentation presents a new algorithm that first detects the potential deadlock pattern instances and then detects a real deadlock by pruning from these instances. The key insight in this presentation is the abstract machine that prunes provably non-feasible schedules by simply counting the issued sends and issued receives. This presentation further defines two types of deadlock patterns: circular dependency and orphaned receive, and their validation. The circular dependency uses simple rules for validation. The orphaned receive, however, requires a higher cost SMT encoding from existing work. This presentation additionally proves that the algorithm is sound and complete for the circular dependency pattern and the orphaned receive pattern. Further, the algorithm can be modified to check deadlocks for these two patterns under zero buffer semantics. Experiments demonstrate that the algorithm

in this presentation is able to detect all the deadlocks in a typical set of benchmarks and is more efficient than two state-of-art MPI verifiers.

It is believed that there exist deadlock patterns other than the circular dependency and the orphaned receive in MPI programs. Future work will define other deadlock patterns and to detect deadlocks for these patterns. Also, future work will explore how to detect deadlocks for programs with branching.

The collective semantics in MPI programs include two essential aspects: internal message communication that is not interrupted by the regular point-to-point communication and capability of program synchronization. The program synchronization can be defined as the behavior of barriers in this paper. The message communication in collective operations, can be modeled by converting them to special point-to-point operations with tags in future work.

Another promising avenue of research is identifying independently verifiable execution paths. Many programs contain sections where all schedules must pass through a shared exit state. This can allow verification algorithm to consider the program as two smaller programs, greatly increasing efficiency.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Message passing models are widely used in communication between multi-core devices applied to high performance computing. Verifying message passing programs is difficult because of two common problems, message-race and deadlock. Detecting their existence in a message passing program is NP–complete. Also, the difficulty of verification comes from three factors that complicate the message passing semantics: asynchronous communication may order the communication primitives in different ways; barrier synchronization may not block a message to move across barriers; and two buffering semantics with different program behaviors are both allowed by the message passing standards. These factors are defined in two prevalent message passing standards: MCAPI and MPI. Therefore, the work in this dissertation applies to these two standards.

The work in this dissertation uses predictive analysis to check program properties. The novelty is not predicting the program behavior in a single execution, but expanding the prediction to a set of executions that arise from the initial execution. The predictive analysis in this research is accomplished in several steps that are presented in Chapter 2 to Chapter 5.

Chapter 2 provides an SMT encoding that is able to reason about the precise program behavior in an MCAPI execution. The novelty of the encoding is the direct use of match pair to capture the message communication between sends and receives. The work proves

the NP–completeness of verifying the existence of message-race in message passing programs. Also, the work gives an algorithm that is able to over-approximate the precise match pairs in an execution. This encoding uses fewer clauses than the prior SMT/SAT encodings [17, 18].

The precise SMT encoding in Chapter 2 is also expanded to MPI semantics in Chapter 3, including point-to-point communication and barrier. The extended encoding is capable of detecting zero buffer incompatibility for MPI programs. The key insight is that the encoding considers only the schedules that strictly alternate sends and receives. Experiments indicate that the work is correct and more efficient than two state-of-the-art MPI verifiers, MOPPER and ISP [20, 49, 57, 59].

The SMT encoding is not going to be scalable. As such, this research gives a heuristic search in Chapter 4 to efficiently reduce the size of the match set as input to the SMT encoding, which thereby decreases the verification cost. The novelty of the heuristic search is process sectioning where sends and receives from different sections can not match. As such, only a subset of the search space is considered. The match generation algorithm in Chapter 2 is then applied for each section. Given the reduced match set generated by the heuristic search, the runtime of the SMT encoding is highly improved compared with that directly using the precise match set.

In addition to the precise SMT based model checking with heuristic search, Chapter 5 provides another scalable approach that uses static analysis to detect deadlocks in MPI programs. The work first detects all the potential deadlocks in a program by statically searching the instances of the pre-defined deadlock patterns. It then prunes the non-feasible deadlocks by an abstract machine based on counting. Finally, the approach validates each remaining deadlock precisely. The validated deadlock schedule represents a real deadlock in the runtime. The work defines two patterns for deadlock: circular dependency and orphaned receive. Experiments show that the work is more efficient than two MPI verifiers and the two deadlock patterns cover all the deadlock cases in the benchmarks.

Given the precise SMT based model checking with heuristic search and the static analysis, the work in this dissertation is able to verify the existence of message-race and/or deadlocks in message passing programs.

6.2 Future Work

The immediate step in the future work is to implement the technique in this dissertation so that it can be used by non-experts. The implementation is able to efficiently detect errors or to witness the correctness in message passing programs.

A further step is to extend the work to support the full set of the MPI standard. For example, the tags defined in point-to-point communication may also cause message non-determinism, and therefore, make the semantics more complex and difficult to reason about. Extending tags to the technique is helpful to find more hidden errors in message passing programs. The extension to tags is also useful to model the full set of collective communication where the messages within collective operations are delivered with special tags. Also, the MPI standard supports other strategies such as one-sided communication and hybrid programming with threads. Future work will extend the technique to these areas. This may require both the new techniques with static analysis, dynamic analysis, and model checking, etc, and the implementation to support our model.

The technique currently works well for the intermediate abstraction of a message passing program – concurrent trace program. This abstraction, however, may not capture all the program behavior from multiple branches, loops, function calls, different data structures or various concrete inputs, etc. that exist in the original program but are not included in the abstraction. Future work needs to extend the technique for concurrent trace programs to real message passing programs. For example, bounded model checking can be used to unroll loops and to explore all the paths. Also, a match pair has to record a feasible path where the match pair may occur at runtime.

Last but not least, the technique has to be improved. The rank based match pair generation algorithm in this dissertation is able to prune a large number of obvious non-feasible match pairs, however, it is not precise. Future work will find more precise and efficient algorithms for match pair generation. It is believed that there exist deadlock patterns other than the circular dependency and the orphaned receive in message passing programs. Future work will define other deadlock patterns and detect deadlocks for these patterns. The work in Chapter 4 gives a feasible direction in verifying message passing programs, where the program is divided such that small problem instances are solved independently. The future work will extend this idea to dividing the program into small programs and verifying them one by one. This strategy may greatly increase efficiency.

References

[1] FEVS benchmark. *http://vsl.cis.udel.edu/fevs/index.html*.

[2] The Yices SMT solver. URL `http://yices.csl.sri.com/`.

[3] *MPI: A Message-Passing Interface Standard Version 3.1*. Message Passing Interface Forum, 2015. URL `http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[4] Clark Barrett, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (smt-lib). *www.SMT-LIB.org*, 2008.

[5] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. In Cormac Flanagan and Barbara König, editors, *TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2012. ISBN 978-3-642-28755-8.

[6] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*. IEEE Computer Society, 2009.

[7] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 12–21. ACM, 2007. ISBN 978-1-59593-633-2.

[8] J. Burkardt. MPI Examples. URL `http://people.sc.fsu.edu/~jburkardt/cpp_src/mpi/mpi.html`.

[9] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence and consistency. *IEEE Trans. Parallel Distrib. Syst.*, 16(7):663–671, 2005. doi: 10.1109/TPDS.2005.86. URL `http://doi.ieeecomputersociety.org/10.1109/TPDS.2005.86`.

[10] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *DAC*, pages 368–371. ACM, 2003. ISBN 1-58113-688-9.

[11] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. ISBN 3-540-21299-X.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi: 10.1145/115372.115320. URL `http://doi.acm.org/10.1145/115372.115320`.

[13] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963, pages 337–340. Springer, Heidelberg, 2008.

[14] Brian Demsky and Patrick Lam. Satcheck: Sat-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 20–36, 2015. doi: 10.1145/2814270.2814297. URL `http://doi.acm.org/10.1145/2814270.2814297`.

[15] Jori Dubrovin, Tommi A. Junttila, and Keijo Heljanko. Exploiting step semantics for efficient bounded model checking of asynchronous systems. *Sci. Comput. Program.*, 77 (10-11):1095–1121, 2012.

[16] Bruno Dutertre and de Moura Leonardo. A fast linear-arithmetic solver for DPLL(T). In *CAV*, volume 4144 of LNCS, pages 81–94. Springer-Verlag, 2006.

[17] Mohamed Elwakil and Zijiang Yang. Debugging support tool for MCAPI applications. In João Lourenço, editor, *PDATAD*, pages 20–25. ACM, 2010. ISBN 978-1-4503-0136-7.

[18] Mohamed Elwakil, Zijiang Yang, and Liqiang Wang. CRI: Symbolic debugger for MCAPI applications. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *ATVA*, volume 6252 of *Lecture Notes in Computer Science*, pages 353–358. Springer, 2010. ISBN 978-3-642-15642-7.

[19] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040315. URL `http://doi.acm.org/10.1145/1040305.1040315`.

[20] Vojtech Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 263–278, 2014. doi: 10.1007/978-3-319-06410-9_19. URL `http://dx.doi.org/10.1007/978-3-319-06410-9_19`.

[21] Ian Gray and Neil C. Audsley. Targeting complex embedded architectures by combining the multicore communications API (mcapi) with compile-time virtualisation. In *Proceedings of the ACM SIGPLAN/SIGBED 2011 conference on Languages, compilers, and tools for embedded systems, LCTES 2011, Chicago, IL, USA, April 11-14, 2011*, pages 51–60, 2011. doi: 10.1145/1967677.1967685. URL `http://doi.acm.org/10.1145/1967677.1967685`.

[22] W. Haque. Concurrent deadlock detection in parallel programs. *Int. J. Comput. Appl.*, 28(1):19–25, January 2006. ISSN 1206-212X. URL `http://dl.acm.org/citation.cfm?id=1165452.1165455`.

[23] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. MUST: A scalable approach to runtime error detection in MPI programs. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 53–66, 2009. doi: 10.1007/978-3-642-11261-4_5. URL `http://dx.doi.org/10.1007/978-3-642-11261-4_5`.

[24] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5): 279–295, 1997.

[25] Yu Huang and Eric Mercer. Detecting MPI zero buffer incompatibility by SMT encoding. *NFM*, 2015.

[26] Yu Huang, Eric Mercer, and Josh Asplund. A hybrid approach of dynamic and static analyses for a hybrid approach of dynamic and static analyses for deadlock in MPI programs. Draft, . URL `http://students.cs.byu.edu/~yhuang2/downloads/deadlock.pdf`.

[27] Yu Huang, Eric Mercer, and Jay McCarthy. Proving MCAPI executions are correct using SMT (extended), . URL `http://students.cs.byu.edu/~yhuang2/downloads\/paper.pdf`.

[28] Yu Huang, Eric Mercer, and Jay McCarthy. Proving MCAPI executions are correct using SMT. In *ASE*, pages 26–36, 2013.

[29] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975. doi: 10.1137/0204007. URL `http://dx.doi.org/10.1137/0204007`.

[30] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 110–120. ACM, 2009. ISBN 978-1-60558-392-1.

[31] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 398–413. Springer, 2009. ISBN 978-3-642-02657-7.

[32] Bettina Krammer, Katrin Bidmon, Matthias S. Müller, and Michael M. Resch. MARMOT: an MPI analysis and checking tool. In *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, pages 493–500, 2003.

[33] Shuvendu K. Lahiri. Smt-based modular analysis of sequential systems code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2011. ISBN 978-3-642-22109-5. doi: 10.1007/978-3-642-22110-1_3. URL `http://dx.doi.org/10.1007/978-3-642-22110-1_3`.

[34] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 171–182, 2008. doi: 10.1145/1328438.1328461. URL `http://doi.acm.org/10.1145/1328438.1328461`.

[35] Glenn R. Luecke, Hua Chen, James Coyle, Jim Hoekstra, Marina Kraeva, and Yan Zou. MPI-CHECK: a tool for checking fortran 90 MPI programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003. doi: 10.1002/cpe.705. URL `http://dx.doi.org/10.1002/cpe.705`.

[36] MCA. The multicore association, . URL `http://www.multicore-association.org`.

[37] MCA. The multicore association resource management API. *http://www.multicore-association.org/workgroup/mcapi.php*, .

[38] Everett Morse, Nick Vrvilo, Eric Mercer, and Jay McCarthy. Modeling asynchronous message passing for C programs. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, pages 332–347, 2012. doi: 10.1007/978-3-642-27940-9_22. URL `http://dx.doi.org/10.1007/978-3-642-27940-9_22`.

[39] MPICH. High-Performance Portable MPI. *http://www.mpich.org*.

[40] MPPTest. MPPTest benchmark. URL `http://www.mcs.anl.gov/research/projects/mpi/mpptest/`.

[41] N. Natarajan. A distributed algorithm for detecting communication deadlocks. In *Foundations of Software Technology and Theoretical Computer Science, Fourth Conference, Bangalore, India, December 13-15, 1984, Proceedings*, pages 119–135, 1984. doi: 10.1007/3-540-13883-8_68. URL `http://dx.doi.org/10.1007/3-540-13883-8_68`.

[42] Robert Netzer, Timothy Brennan, and Suresh Damodaran-Kamal. Debugging race conditions in message-passing programs. In *ACM SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 31–40, Philadelphia, PA, USA, 1996.

[43] Mi-Young Park, Su Jeong Shim, Yong-Kee Jun, and Hyuk-Ro Park. Mpirace-check: Detection of message races in MPI programs. In *Advances in Grid and Pervasive Computing, Second International Conference, GPC 2007, Paris, France, May 2-4, 2007, Proceedings*, pages 322–333, 2007. doi: 10.1007/978-3-540-72360-8_28. URL `http://dx.doi.org/10.1007/978-3-540-72360-8_28`.

[44] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *CAV*, pages 82–97, 2005.

[45] Subodh Sharma. Private conversation on active research.

[46] Subodh Sharma. *Ph.D. Dissertation: Predictive Analysis of Message Passing Applications*. The University of Utah, March 2012.

[47] Subodh Sharma, Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. A formal approach to detect functionally irrelevant barriers in mpi programs. In Alexey L. Lastovetsky, M. Tahar Kechadi, and Jack

Dongarra, editors, *PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, pages 265–273. Springer, 2008. ISBN 978-3-540-87474-4.

[48] Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. MCC: A runtime verification tool for MCAPI user applications. In *FMCAD*, pages 41–44. IEEE, 2009. ISBN 978-1-4244-4966-8.

[49] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. A sound reduction of persistent-sets for deadlock detection in MPI applications. In *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*, pages 194–209, 2012. doi: 10.1007/978-3-642-33296-8_15. URL `http://dx.doi.org/10.1007/978-3-642-33296-8_15`.

[50] Subodh Sharma, Ganesh Gopalakrishnan, and Greg Bronevetsky. Abstract: MAPPED: predictive dynamic analysis tool for MPI applications. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, pages 1425–1426, 2012. doi: 10.1109/SC.Companion.2012.233. URL `http://dx.doi.org/10.1109/SC.Companion.2012.233`.

[51] Stephen F. Siegel. Verifying parallel programs with MPI-Spin. In *PVM/MPI*, pages 13–14, 2007.

[52] Stephen F. Siegel. Model checking nonblocking MPI programs. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 44–58. Springer, 2007. ISBN 978-3-540-69735-0.

[53] Stephen F. Siegel and Ganesh Gopalakrishnan. Formal analysis of message passing - (invited talk). In Ranjit Jhala and David A. Schmidt, editors, *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2011. ISBN 978-3-642-18274-7.

[54] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of mpi-based parallel programs. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 309–310, 2011. doi: 10.1145/1941553.1941603. URL `http://doi.acm.org/10.1145/1941553.1941603`.

[55] Stephen F. Siegel and Timothy K. Zirkel. Loop invariant symbolic execution for parallel programs. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24,*

2012. Proceedings, pages 412–427, 2012. doi: 10.1007/978-3-642-27940-9_27. URL http://dx.doi.org/10.1007/978-3-642-27940-9_27.

[56] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. CIVL: the concurrency intermediate verification language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, pages 61:1–61:12, 2015. doi: 10.1145/2807591.2807635. URL http://doi.acm.org/10.1145/2807591.2807635.

[57] Sarvani S. Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *CAV*, pages 66–79, 2008.

[58] Sarvani S. Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: a tool for model checking MPI programs. In *PPOPP*, pages 285–286, 2008.

[59] Sarvani S. Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. Reduced execution semantics of MPI: From theory to practice. In *FM*, pages 724–740, 2009.

[60] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with umpire. In *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, page 51, 2000. doi: 10.1109/SC.2000.10055. URL http://doi.ieeecomputersociety.org/10.1109/SC.2000.10055.

[61] Anh Vo, Sarvani S. Vakkalanka, and Ganesh Gopalakrishnan. ISP tool update: Scalable MPI verification. In *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, pages 175–184, 2009. doi: 10.1007/978-3-642-11261-4_12. URL http://dx.doi.org/10.1007/978-3-642-11261-4_12.

[62] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *SC*, pages 1–10. IEEE, 2010. ISBN 978-1-4244-7559-9.

[63] Anh Vo, Ganesh Gopalakrishnan, Robert M. Kirby, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. Large scale verification of MPI programs using Lamport clocks with lazy update. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 330–339. IEEE Computer Society, 2011. ISBN 978-1-4577-1794-9.

[64] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 23–32. ACM, 2009. ISBN 978-1-60558-001-2.

[65] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey M. Voelker. MPIWiz: subgroup reproducible replay of mpi applications. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 251–260, 2009. doi: 10.1145/1504176.1504213. URL `http://doi.acm.org/10.1145/1504176.1504213`.

[66] Yuan Zhang and Evelyn Duesterwald. Barrier matching for programs with textually unaligned barriers. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *PPOPP*, pages 194–204. ACM, 2007. ISBN 978-1-59593-602-8.

[67] Manchun Zheng, Michael S. Rogers, Ziqing Luo, Matthew B. Dwyer, and Stephen F. Siegel. CIVL: formal verification of parallel programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 830–835, 2015. doi: 10.1109/ASE.2015.99. URL `http://dx.doi.org/10.1109/ASE.2015.99`.