Experience Report: Verifying MPI Java Programs using Software Model Checking

Muhammad Sohaib Ayub*, Waqas Ur Rehman†, Junaid Haroon Siddiqui‡
*‡Department of Computer Science, LUMS School of Science and Engineering, Lahore, Pakistan
†Department of Computer Science, University of Saskatchewan, Saskatoon, Canada
*15030039@lums.edu.pk,†wur218@mail.usask.ca,‡junaid.siddiqui@lums.edu.pk

***Abstract*—Parallel and distributed computing have enabled development of much more scalable software. However, developing concurrent software requires the programmer to be aware of non-determinism, data races, and deadlocks. MPI (message passing interface) is a popular standard for writing message-oriented distributed applications. Some messages in MPI systems can be processed by one of the many machines and in many possible orders. This non-determinism can affect the result of an MPI application. The alternate results may or may not be correct. To verify MPI applications, we need to check all these possible orderings and use an application specific oracle to decide if these orderings give correct output.**

MPJ Express is an open source Java implementation of the MPI standard. Model checking of MPI Java programs is a challenging task due to their parallel nature. We developed a Java based model of MPJ Express, where processes are modeled as threads, and which can run unmodified MPI Java programs on a single system. This model enabled us to adapt the Java PathFinder explicit state software model checker (JPF) using a custom listener to verify our model running real MPI Java programs. The evaluation of our approach shows that model checking reveals incorrect system behavior that results in very intricate message orderings.

I. Introduction

Model checking [6] is a powerful program analysis technique based on systematic exploration of nondeterministic choices in a program. Nondeterministic choices could be program inputs, modeled behavior of an external system, thread interleavings in a multithreaded program, or even message orders in a distributed system. Model checking has been used to verify models of hardware circuits [5]. Earlier software model checkers required converting the program into a modeling language which was then verified for certain properties [6]. Recent software model checkers, such as the Java PathFinder (JPF) [14], now provide the foundation of an increasingly effective toolset for systematic checking of programs written in commonly used languages. Recent technological advances have well-supported the core model checking techniques, and software model checkers are being applied to larger and larger programs.

Despite the progress, adapting the model checking techniques for new domains remains a challenging problem in realizing its true potential in increasing our ability to deploy more reliable software systems. There are three primary causes for that. One is to adapt the program under test to run in the limitations of a given model checkers (e.g., library support, source code requirements, single machine support, etc.). Second is to enable monitoring and exploration of new sources of non-determinism, and third is to allow partial order reduction (POR) to work effectively. POR [9] is the mechanism model checking uses to reduce the number of choices explored on the basis of equivalence sets.

One such domain is MPI programs. Message passing interface (MPI) is a framework for the development of parallel and distributed applications. It has been implemented in C, FORTRAN, and Java. The Java implementation is called MPJ Express[1]. It is deployed on different machines communicating through an underlying communication channel. MPI programs often form the basis of large distributed systems, and their correctness is extremely critical. However, testing them is difficult because there are many possible orders messages can arrive in and it is hard to find if any one of these message orders does not give the desired outcome.

This paper presents a novel technique to adapt model checking for unchanged MPI programs written in MPI Java. While prior work addressed this problem requiring conversion of the MPI program into modeling languages [26] or provided general guidelines of how such a solution can be built [10], we provide a concrete end to end implementation to model check unchanged MPI Java Programs. Our choice of using MPI Java enables us to adapt the existing Java PathFinder model checker, but the fundamental techniques are not specific to Java.

Designing and Debugging of MPI applications is difficult due to their parallel nature causing issues like non-determinism, race conditions, deadlocks, etc. Asynchronous communication and difficulty to produce global state due to distributed environment create additional challenges. Further, faults in distributed applications can occur at different components and at different layers which make testing of such application more complex. In this work, we explore all possible interleavings which highlight the non-deterministic nature of the application. We make the model of the complete system which ensures loss-free communication for the testing environment in addition to the global visibility of complete state of the system. Further, we do not require injecting the testing/simulation mechanism at different layers due to the generation of the complete model of the system.

Our key insight is that an MPI Java proxy model which converts the new source of non-determinism, i.e. message

[1]http://mpj-express.org/

2332-6549/17 $31.00 © 2017 IEEE
DOI 10.1109/ISSRE.2017.15

orderings, into events that the existing JPF model checker can understand, enables model checking of unchanged MPI Java programs. Such a model also enables effective partial order reduction (POR) which is built in Java PathFinder. POR is a technique to reduce state space exploration by detecting execution of the same state in different order. Such MPI Jave proxy model alone is not enough for JPF to model check MPI applications. JPF does not understand Send/Receive calls or message orders. We have used JPF as a platform and extended it to explore the state of messages. Further complications are added by non-blocking receivers e.g. a non-blocking receive has to be temporarily blocked so other MPI processes can send before it moves on to ensure that all message orders are exercised. To implement this, we introduced a custom listener for message related events, a custom choice point to explore alternate message orderings and a custom MPJ Express proxy that translated calls for the unchanged program.

We make the following contributions:

- **Model checking of MPI Java programs.** We demonstrate an end-to-end technique for model checking of unchanged MPI Java programs using the Java PathFinder model checker.
- **MPJ Express proxy model.** We defined semantics for an MPJ Express proxy model that behaves like the native library (which is not supported in JPF) and simulates processes in threads and message passing in queues whose order can be rearranged.
- **JPF Adaptation.** We adapted JPF for model checking MPI Java programs using a custom listener extension in JPF. One key technique we use to explore all message orders is sender priority. Our receivers block until no senders are left in the system. This blocking enables all message choices to accumulate in the queues. Sender blocking is only made possible due to all states being centrally available in the model checker.
- **Implementation.** We provide a complete end-to-end implementation of adapted Java PathFinder model checker that can model check unchanged MPI Java programs. The implementation is specific to MPI Java and JPF, but the techniques can be adapted to MPI programs in other languages or even other message based systems not using MPI.
- **Evaluation.** We evaluated our technique on a number of small programs and showed that we could effectively explore message orders and find cases where the answer differs based on message orders. We have also discussed scalability of our approach.

The key idea of the paper was earlier presented in a poster paper in the Proceedings of 21^{st} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2016) [22]. The current paper expands the idea presented in that earlier work with detailed algorithms, implementation, and evaluation.

Rest of the paper is structured as follows: Section II discusses about model checking, model checker Java Pathfinder and Message passing interface in Java. Technique and implementation details of our work are discussed in Section III and IV respectively. Section V explains evaluation of our work and related work is discussed in Section VI. We conclude presented work in section VII.

II. Background

In this section, we will describe model checking with its benefits and limitations, introduce Java PathFinder (JPF) and MPJ (Message Passing Interface in JAVA).

A. Model Checking

Ensuring software and hardware correctness is an important issue as failure of either can result in huge financial loss and may have fatal consequences in safety critical systems. Initially model checking [28] was used for hardware verification only, but later software verification research community also adopted the technique.

Model checking, provided a model of system, automatically checks if the system meets given specifications. This checking should be exhaustive. Here, specifications include functional as well as safety requirements. Safety requirements cover absence of deadlocks and race conditions and any other state that can cause the system to crash.

We define state as the contents of memory and processor registers at any given time. State space is all the possible states of the program. State explosion is the main challenge faced by model checking techniques. In any system, when the number of variables increases, the possible states of the system increase exponentially. This exponential increase results in state space explosion problem. Model checking works only for those systems, which either have finite states or finite state abstractions, otherwise model checking fails. Model checking tools use Partial Order Reduction (POR) technique to cater for the issue of state explosion. POR records all the states created and whenever a new state is generated, it maps this new state to existing ones and if matches, it stops as this is an already explored path. Using POR mechanism, we can reduce the chances of state explosion but cannot eliminate it completely.

B. Java PathFinder (JPF)

Java PathFinder (JPF) [14] is a software model checker for Java programs which does not require the specification to be written in a separate language. JPF core is the Virtual Machine(VM) for Java bytecode which means you can execute actual Java code on it. It executes Java code to find defects and give assurance whether the given properties hold or not.

Software model checking is all about making the right choices. The most useful feature of JPF is its capability to identify the points in a program with different execution choices and explore all of them using choice generators, called scheduling choice generators. There are many types of scheduling choice generators, for example, if two threads want to access a single variable, then a "SharedChoiceGenerator" is created. Whenever JPF starts analysis of any program, it

creates a "RootChoiceGenerator" which is essential for JPF model checking. Theoretically, JPF can explore all possible paths through the program, unlike a normal VM. Other than scheduling choice generators, JPF provides data choice generators which provide capability to explore the same execution path with set of different data values.

We cannot ignore the fact that exploring all paths could result in state space explosion. Partial Order Reduction (POR) is one of the main technique to minimize the effect of state explosion problem in model checking. In POR, system compares the states with already visited states and chooses a path which is not yet explored. Each time JPF reaches a point where more than one execution is possible, it performs state matching of current state with previously visited states. If current state is already visited, it backtracks and chooses unexplored paths. JPF performs POR during model checking of any program.

JPF has an important extension mechanism called listeners. Using listeners, we can observe, interact and extend JPF. We can observe current state id, instruction or choice generator in execution, state of each thread and many other events. In our work, we have extended the JPF using listener for controlling and scheduling of thread interleaving.

As explained earlier, JPF itself is a VM, which means it executes the Java code as JDK. When JPF runs a Java code, converted to bytecode, it needs the implementation detail of each bytecode instruction. JPF has complete bytecode instruction factory which contains all the bytecode instruction classes. JPF uses these classes when running Java code.

C. Message Passing Interface in Java (MPJ)

Parallel computing widely uses message passing interface for communication. MPI standard was proposed by High-Performance Community to avoid vendor specific implementation of message passing interface. It defines user interfaces and functionality for wide range of message-passing capabilities. The major goal of MPI, as most standards have, is the degree of portability across different machines. By different machines, we mean running MPI standard transparently on heterogeneous systems [11]. Traditionally library implementations for MPI standard were developed for C/C++ and Fortran languages because C and Fortran bindings were defined by MPI Standard Document.

For Java HPC community, Java binding for MPI Standard was suggested which is called MPJ API or simply MPJ [1]. MPJ Express [2] is a software which provides reference implementation of MPJ API for writing MPI Java programs. MPJ Express completely implements MPJ standard. It supports all primitive and user-defined data types. It allows point-to-point and collective communication with the flexibility of synchronous and asynchronous operations. A simple MPI Java program in which one process sends messages to two other processes is shown in Figure 1.

```
import mpi.*;
public class Main {
 public void main(String [] args) {
  MPI.Init(args) ;
  if(rank == 0) {
   String buffer[] = new String[2] ;
   buffer[0] = "1,2,3,4" ;
   buffer[1] = "5,6,7,8" ;
   for(int i = 1 ; i <= 2 ; i++) {
    MPI.COMM_WORLD.Send(buffer, (i-1),
    1, MPI.OBJECT, i, 100) ;
   }
  } else {
   String buffer1[] = new String[1] ;
   Status status =
       MPI.COMM_WORLD.Recv(buffer1, 0,
    buffer1.length, MPI.OBJECT, 0, 100);
   }
   MPI.Finalize() ;
  }
 }
```

Fig. 1: Simple MPI Java Program

III. Our MPJ Model

MPJ Express is a framework to run MPI Java programs in a cluster as well as in a multicore environment. Each MPJ process is identified by a unique rank number, assigned by the framework. MPJ Express provides built-in functions Rank(), Size(), Recv() and Send() to get the rank of specific process, total number of processes in communication, receiving a message, and sending a message. We are using Java PathFinder (JPF) as model checker for our work. JPF has a limitation that it cannot model check programs executing on different machines. To cater for this issue, we have developed our model of MPJ Express, shown in Figure 2. Our model uses multithreading in which each thread represents a real MPI Java process, as shown in Figure 3. An MPI process (aka distributed node) is usually mapped to an OS process whereas our technique maps it to an OS thread. This enables us to model distributed communication using queues and to use JPF to model check the single process containing all these threads. For each MPJ process, we have created a thread and assigned it a rank number, same as its rank in MPJ Express. Each MPJ process, modeled as thread, can access its rank number in the same manner as in the MPJ Express by calling Rank() function. We have also modeled Size() function which returns the total number of threads executing. "MPJ_Express_Model" is our configuration and thread creating class. Our model configures the number of threads in "MPJ_Express_Model" class, which is given as the number of processes parameter to MPI Java programs. Our model uses Java reflection for running unmodified MPI Java programs. Java reflection enables us to execute multiple threads of an unmodified MPJ program. Intracomm class in MPJ Express framework is responsible for message sending and receiving functionality. We have also implemented it with identical functionality for our model.

Real MPI Java programs send and receive messages using the underlying physical infrastructure for communication. In our model, we are using a shared synchronized queue to

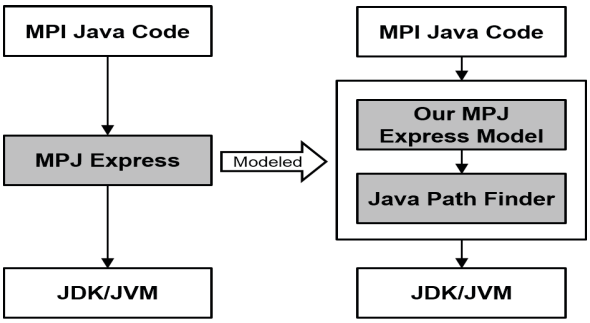


Fig. 2: Mapping of MPJ Express Programs to Our Model

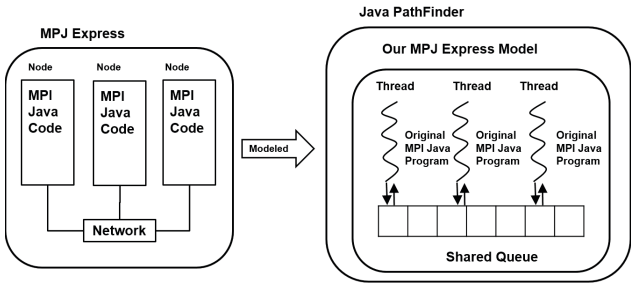


Fig. 3: Our MPJ Express Model

represent physical communication channel, as shown in Figure 3. This shared queue represents the underlying physical infrastructure used by real MPI Java programs. Using a shared synchronized queue to model communication channels ensures prevention of deadlock or race condition. Each enqueue operation represents send message function of the real system. In the same way, each dequeue operation represents receive message function of the real system. Threads enqueue and dequeue messages in the queue to exhibit Send() and Recv() behaviors respectively. In send function, we enqueue the message with send function arguments containing receiver rank and tag. We are using 100 as the special sender rank to specify wildcard receive. For receive function, the Wait Notify mechanism of Java is used if message is not in the queue otherwise message is dequeued from the queue. Messages in our model can not be dropped due to shared queue which implies completely loss-free communication channel. Our MPJ Express model supports all data types which are supported by MPJ Express. Our technique can also perform deadlock detection for MPI programs. Once the MPI application is modeled in JPF and all message orders are exercised, JPF will find message orders that result in deadlocks for free. Partial order reduction (POR) mechanism of JPF effectively prunes the state space of our model checking process.

Our proposed technique is sound because if it finds any message interleaving producing incorrect result, it is guaranteed that same interleaving can be produced by one of the possible message orderings of the original MPI Java program. When a sender sends a message that more than one receiver can receive, we exercise all scenarios where the message is received by each of the different receivers. Thus every interleaving found is a possible and valid interleaving. Our approach is not complete because if it does not find any interleaving producing incorrect results, then it might be due to some limitation of model checking e.g. state space explosion.

IV. Implementation Details

Existing model checkers (e.g. Java PathFinder) cannot model check MPI Java programs directly because such programs are running on different machines as well as due to MPJ Express's dependency on native Java libraries. So, we have implemented our proposed technique to model MPI Java programs without any modifications in source code or writing additional specifications. We have created our own listener and package of supporting classes for logging in JPF. In this section, we will explain implementation details of our technique for model checking of MPI Java (MPJ) programs.

A. Supporting Classes

JPF is extended with a listener and a set of supporting classes to perform model checking of MPJ programs. These supporting classes are used to monitor and for logging state of the model checking processes. StateNode is the basic class for logging state variables corresponding to each state in JPF, as shown in Figure 4. Additionally, it logs the blocked threads in current state as well as from immediate parent to root node. It also holds the information of threads which dequeued at least one message and waiting for another message. StateTree class is used for logging the overall state space of the system along with blocked threads at each stage, inheriting from parent and further passing to children. It initiates new state by inheriting the blocked threads of parent and logging those threads that already got at least one message from the queue and are waiting for more messages. To block execution of a thread if there is no message for the respective thread, WaitingListClass is used to get the updated status of queue. Queue status is refreshed whenever a new message is inserted, or thread is going to get the second message out of queue. The update on second message ensures that the previous message for the same thread has been removed from the queue.

B. Listener for State Space Exploration

In the start of the program, all runnable threads are either senders or receivers. Threads want to send the message(s) put the message(s) in the common queue, MessageQueue data structure in our implementation as shown in Algorithm 1. If these threads have successfully put their message(s), they get terminated, or if same thread is expecting to receive the message(s) in the future, that threads get blocked. Threads which start with receiving messages also get blocked at the start for interleaving exploration. Blocking or termination of thread will continue until either all threads get terminated after sending messages or get blocked. All blocked and terminated

```
// StateNode.java
package gov.nasa.jpf.supportingClasses;
import java.util.ArrayList;
import gov.nasa.jpf.vm.ThreadInfo;
public class StateNode {
 public int stateId;
 public StateNode Parent;
 public ArrayList<StateNode> Children;
 public ArrayList<ThreadInfo> Threads;
 public ArrayList<ThreadInfo> ParentThreads;
 public ArrayList<ThreadInfo> ThreadLog;
}
// StateTree.java
public class StateTree {
 private static StateNode Root;
 private static StateNode CurrentNode;
 public StateTree() {...}
 public void StateAdvance(int _state_id,
 ArrayList<ThreadInfo> _threadLog) {...}
 /* Other Functions */
}
```

Fig. 4: Monitoring Classes

threads are logged by BlockedThreads and TerminatedThreads data structures respectively, as shown in Algorithm 1.

A new choice generator will be created scheduling all blocked threads waiting for messages from the queue, represented as ChoiceGenerator. Each thread (i.e. choice) checks if there is a message for it in queue or not. If there is a message in the queue, it dequeues and proceeds with normal execution. If there is no message for this thread, it gets blocked again. ExecuteThread is the function in mentioned algorithm which is dealing with blocking of threads and controlling enqueue, dequeue operations.

All data structures are empty at the start of the program except Threads and ChoiceGenerator which are initialized with all the runnable threads. The point of execution where all threads are either blocked or terminated, a ChoiceGenerator is created from BlockedThreads. Schedule function is responsible for creating ChoiceGenerator from BlockedThreads.

We are also keeping track of states of our program regarding runnable, blocked and terminated threads. StateStorage data structure is maintaining the log of our all states. After updating the StateStorage, each thread (choice) from ChoiceGenerator get scheduled calling ExecuteThread function. Scheduler function is responsible for tracking the states and scheduling choices in our implementation.

C. Choice Generators

In Java PathFinder (JPF), we have disabled all scheduling choice generators which are normally created by JPF for creating thread interleavings except "TerminatorChoiceGenerator". "TerminatorChoiceGenerator" is responsible for creating thread interleavings when a thread is terminated and helps to explore all possible interleavings. In addition, "RootChoiceGenerator" is used to model checking of program by JPF. The

Algorithm 1: Interleaving Exploration Algorithm

```
1  MessageQueue ← Empty
2  StateStorage ← Empty
3  TerminatedThreads ← Empty
4  BlockedThreads ← Empty
5  Threads ← AllRunnableThreads()
6  ChoiceGenerator ← Choices(Threads)
7  Scheduler(ChoiceGenerator)
8  Function Scheduler(ChoiceGenerator)
9      currentState ← RunnableThreads ∪
       BlockedThreads ∪ TerminatedThreads
10     if currentState ∉ StateStorage then
11         StateStorage ← StateStorage ∪ currentState
12     else
13         return
14     foreach Choice ∈ ChoiceGenerator do
15         ExecuteThread(Choice)
16
17 Function ExecuteThread(thread)
18     if thread.instruction = Recv() then
19         if thread ∈ BlockedThreads then
20             if MessageQueue ∋ MessageForThread then
21                 MessageQueue.Dequeue()
22             else
23                 BlockedThreads.Enqueue(thread)
24                 Schedule(AllThreads \
                   (BlockedThreads ∪ TerminatedThreads))
25         else
26             Schedule(AllThreads \ (BlockedThreads ∪
                   TerminatedThreads))
27     else if thread.instruction = Send() then
28         MessageQueue.Enqueue(thread.message)
29     else
           /* Execute Program Source Code      */
30
31 Function Schedule(Threads)
32     if Threads ≠ ∅ then
33         ChoiceGenerator ← Choices(Threads)
34     else
35         ChoiceGenerator ← Choices(BlockedThreads())
36         BlockedThreads ← ∅
37     Scheduler(ChoiceGenerator)
```

choice generator of "ThreadChoiceFromSet" type is created manually by our listener when all threads get blocked after inserting messages in the queue, and out-of-order reception of messages from the queue is intended. Disabling JPF's ChoiceGenerators and creating own ChoiceGenerator ensure complete control of thread scheduling. At the start of JPF execution, there is only "RootChoiceGenerator" with main thread of our configuration class "MPJ_Express_Model". As a result of termination of main thread, a "TerminatorChoiceGenerator" is created with all runnable threads as choices. JPF makes choice of one runnable thread for execution while the listener observes the execution.

D. Thread Interleaving Exploration

Threads in our model can have two states, which are blocked state and unblocked state. Threads waiting for the message(s) in queue or from other threads are said to be in blocked state. Unblocked (runnable) threads are those threads which are never executed from the start of program or unblocked after the creation of choice generator to execute all the blocked threads.

Threads can be categorized into three types according to their send/receive behavior i.e. (*Type I*) thread only sends message(s), (*Type II*) thread only receives message(s) and (*Type III*) thread both sends and receives message(s).

During the execution of *Type I* thread, it enqueues messages in shared queue and terminates. As a result, a "TerminatorChoiceGenerator" is created with all runnable threads which is further used to explore all interleavings. We have instrumented JPF in such a way that whenever a thread reaches Recv() function, JPF pause the execution of respective thread, creates a new choice generator of "ThreadChoiceGenerator" type and schedule remaining unblocked threads. So, *Type II* threads get blocked when their execution reaches at Recv() function. It is quite possible that this thread does not get all messages from queue in its first attempt, it remains blocked until some message is found in queue for this thread. We have implemented in our listener that state of the queue is refreshed whenever a send function call returns or a thread tries to get message from queue. Thread terminates after getting all the messages from the queue. Processing of *Type III* thread is basically the combination of both *Type I* and *Type II* threads. If a thread sends message followed by receive and message to be received is not found in the queue, it starts waiting until all other threads either get blocked or terminate. If it receives message followed by send, it gets blocked to receive message until all remaining threads either get blocked or terminated. Once it receives the message, it enqueues message to send in queue.

A thread gets blocked at Recv() function call because we want all threads to send messages first to explore all the interleaving. When all runnable threads get blocked, choice generator is created to schedule all blocked threads. When a thread gets blocked during the execution, a new choice generator is created with remaining runnable threads, and choice generator explores all the choices one by one. Blocked threads are logged in StateTree class, and their state is logged in StateNode structure. When another thread gets blocked, choice generator is again created with remaining runnable threads. These new lower level choice points need to consider previously blocked threads as well. So, StateNode holds all the blocked threads log from current node's parent to root node. In case of termination of a thread, "TerminatorChoiceGenerator" is created with all the runnable threads including blocked threads. So, we have modified JPF's "ChoiceGeneratorAdvance" function, which explores new choices, to process unblocked choices ignoring blocked ones. In this way, we ensure all the threads are executed in all possible interleaving and find any interleaving producing incorrect result due to non-determinism in the given MPJ program.

V. Evaluation

We have implemented our technique in JPF by extending it with custom listener and making other necessary modifications, as described earlier. We have evaluated our technique to model check MPI Java programs and compared its performance with MPJ Express. We have used order-sensitive distributed applications using point-to-point communication to evaluate our work. Different interleaving of processes in order-sensitive applications may produce wrong results. Our work can be used to verify any MPI Java program using point-to-point communication without any code modification or additional dependencies. MPJ processes are modeled as threads in our approach. Therefore, MPJ processes are referred to as threads in this section and 'Rank i' means 'thread with Rank i' for simplicity. We have explained three examples in this section with state diagrams and all possible scheduling options explored by our model. The first example gives the basic understanding of how our system explores all possible states of a program and discard repeated states. The other examples are variants of the running average problem implemented in MPI Java program. Verification of same problem with different algorithm and workload shows that our technique can verify any MPI Java program without need of any modification successfully Running average is used for the analysis of data by calculating average of subsets of data. It is used in many real-world applications, e.g., technical analysis of stock prices, gross domestic product in economics, etc. In case of parallel implementation of the problem, order of the messages containing local average alters the final result of program. Our evaluation shows that our model can detect any incorrect ordering due to non-determinism in message-passing applications.

A. Point-to-Point Communication between 1 Sender and 2 Receivers

The purpose of this example is to explain how JPF explores all possible states of MPI Java program using our MPJ Express model. In this example, three MPI Java threads with Rank 0, Rank 1 and Rank 2 are initiated. Rank 0 is responsible for work distribution. Rank 1 and Rank 2 receive the assigned work from Rank 0. Complete state diagram is also shown in Figure 5. In this example, each thread wants to receive message (using Recv() function call) gets blocked and other runnable threads continue with their execution. We do not stop execution of any thread which wants to send message (Using Send() function call). Message sending threads either get terminated or get blocked on receiving a message after sending messages.

The word "State" is used in little different meanings; a "State" is 3-tuple set containing runnable threads, blocked threads, and terminated threads: State i[(Runnable Threads), (Blocked Threads), (Terminated Threads)]. The order of threads in a tuple does not matter. A state is already visited if

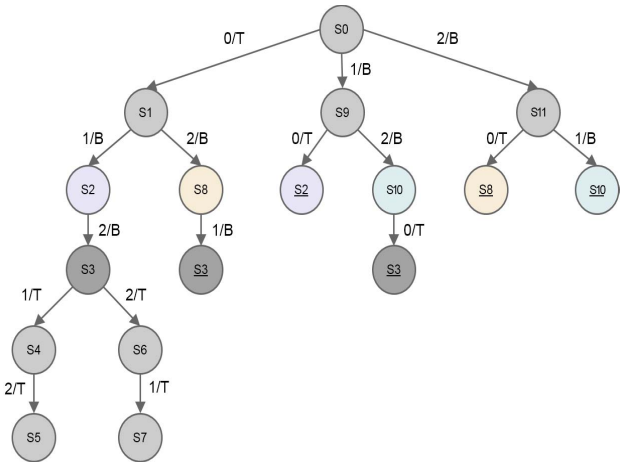


Fig. 5: State Space of Point-to-Point Communication Between 1 Sender and 2 Receivers

TABLE I: Point-to-Point Communication Between 1 Sender and 2 Receivers: State Details - Si[(Runnable Threads), (Blocked Threads), (Terminated Threads)]

Serial No.	State number	State Details	State Type
1.	State 0	S0[(0,1,2), (), ()]	New
2.	State 1	S1[(1,2),(), (0)]	New
3.	State 2	S2[(2), (1), (0)]	New
4.	Transition State 1	TS1[(), (1,2), (0)]	New
5.	State 3	S3[(1,2), (), (0)]	New
6.	State 4	S4[(2), (), (0,1)]	New
7.	State 5	S5[(), (), (0,1,2)]	New
8.	State 6	S6[(1), (), (0,2)]	New
9.	State 7	S7[(), (), (0,1,2)]	New
10.	State 8	S8[(1), (2), (0)]	New
11.	Transition State	TS[(), (1,2), (0)]	TS1
12.	State 9	S9[(0,2), (1), ()]	New
13.	Transition State	TS[(2), (1), (0)]	State 2
14.	State 10	S10[(0), (1,2), ()]	New
15.	Transition State	TS[(), (1,2), (0)]	TS1
16.	State 11	S11[(0,1), (2), ()]	New
17.	Transition State	TS[(1), (2), (0)]	State 8
18.	Transition State	TS[(0), (1,2), ()]	State 10

we have seen the same set before. Table I shows the complete state space of example mentioned above, which is necessary to get the better understanding.

JPF starts model checking process with "RootChoiceGenerator". JPF finds three threads as runnable and generates a choice point with three options i.e. Rank 0, Rank 1 and Rank 2. Starting with Rank 0, it sends messages to other two threads and terminates. At this point, "TerminatorChoiceGenerator" is created (State 1) with remaining runnable threads in system (i.e. Rank 1 and Rank 2).

As the first choice, Rank 1 executes and gets blocked on Recv(). A new choice point (State 2) is generated with all possible runnable threads. Rank 2 executes again and block on Recv(). Now, a state is reached with executable threads as choice. So, we have extended JPF in such a way that it schedule all paused threads(State 3). Transition State 1 (Sr. No 4 in Table I) is not created by JPF, but this represents a state where we do not have any executable threads, and there are threads blocked earlier. In this case, our listener in JPF creates a choice point with all blocked threads.

On State 3, the first choice out of two choices (Rank 1 and Rank 2) runs, receives message and terminates. "TerminatorChoiceGenerator" is created with only Rank 2 as choice (State 4). It executes, receives message and terminates. A new end state (State 5) is generated with no executable threads. At this point, system backtrack to previous states until some state still has choice(s) to be processed. In this case, State 3 has one choice to be processed, backtracking stops there, and Rank 2 executes till termination. As a result, "TerminatorChoiceGenerator" is created (State 6) and runs with all remaining executable threads as choices. After execution of Rank 1, State 7 is reached, which is end state. System backtracks to the previous states and marks them processed until some state with available choice(s) is found. In this case, system backtracks State 6, State 3, State 2 till State 1 and start executing the other choice, i.e. Rank 2. Rank 2 gets blocked on Recv() after creating new choice point (State 8) with Rank 1 as executable. Thread with Rank 1 gets blocked after execution, which leads to the state already seen by JPF (Sr. No. 11). It continues to State 3 which has no choice and backtracks till State 0.

At this point, JPF starts with second choice, i.e. Rank 1. As it reaches Recv() function, it gets blocked, and new choice point is created (State 9) with Rank 0 and Rank 2 as choices. Starting with Rank 0 as choice, it gets blocked after execution leading to already visited state, i.e. State 2 (Sr. No. 12). Upon executing second option Rank 2, it gets blocked, and new choice point is created (State 10) with Rank 0 as executable. After execution of Rank 0 system again reaches already seen state, i.e. State 3 (Sr. No. 15). States are backtracked until State 0 is reached with Rank 2 as available choice. Rank 2 executes, gets blocked at Recv() and new choice point is generated (State 11) with Rank 0 and Rank 1 as executable. The system reaches to already seen state (State 8) after execution of Rank 0 and it reaches to already visited state (State 10) by executing Rank 0.

TABLE II: Roles of Threads

Thread	Responsibilities	Level
Rank 0	Initiator, Terminator	-
Rank 1	Ordinary, Intermediary	1
Rank 2	Ordinary	-
Rank 3	Ordinary, Intermediary	0
Rank 4	Ordinary	-

TABLE III: Intermediary Workers with Responsibilities

Intermediary Worker	Responsible for Ordinary Workers	Level
Rank 3	Rank 4	0
Rank 1	Rank 2, Rank 3	1
Rank 0(Terminator)	Rank 1	-

TABLE IV: Work Distribution by Initiator (Rank 0)

Ordinary Worker	Work	Average
Rank 1	[1,2,3,4]	2.5
Rank 2	[5,6,7,8]	6.5
Rank 3	[9,10,11,12]	10.5
Rank 4	[13,14,15,16]	14.5

B. Running Average with Distributed Receivers

In this section, we are going to present an example of calculating running average of first 16 integers. We have used five threads for this example, each thread having a specific role. Each thread must have at least one of the four roles, that are initiator, intermediary, ordinary and terminator. The initiator is responsible for distributing the work among other threads. Ordinary workers are responsible for processing and forwarding results to intermediary threads. Intermediary threads, as name suggests, are responsible for collecting results from ordinary threads, consolidate them and forward results to either higher level intermediaries or terminator. Terminator collects the final results. Each thread can have more than one role as well. Table II shows the roles of each thread in this example i.e. Rank 0 act as initiator and terminator whereas other threads are ordinary/intermediary workers.

As discussed earlier, our MPJ Express model assigns each thread a unique rank number starting from 0. Rank 0 act as initiator and terminator so it is responsible for distribution of task and collection of final result. Rank 1 act as ordinary and level 1 intermediary worker. It performs assigned task, collects results from other lower level intermediaries (Rank 3) and ordinary workers (Rank 2) and forwards consolidated result to the terminator thread. Rank 3 is level 0 intermediary worker and responsible for collecting results from Rank 4. Thread with the lower level is less responsible for collecting results from other intermediaries/ordinary threads. For example, Level 0 threads are not responsible for collecting results from intermediaries, and they just collect the result of ordinary workers. Rank 2 and Rank 4 are ordinary workers. They calculate the perform the assigned task and forward result to Rank 1 and Rank 3 workers respectively. Table III shows intermediary workers assigned to ordinary workers, as explained above.

Rank 0 distributes work according to Table IV i.e. each worker is assigned four integers for average calculation and waits for the final result from Rank 1 for output. Now, each worker first calculates the average of task assigned to it and forwards the result to respective intermediary thread. Order of messages being received at intermediary nodes is important. Rank 1 distributes array among other threads. Each worker calculates local average of the array assigned to it and start forwarding/receiving result to/from respective worker. Rank 2 and Rank 4 calculate averages of numbers assigned to them and forward messages to Rank 1 and Rank 3 respectively. Upon receiving result from Rank 4, Rank 3 consolidates it with locally calculated average and forward result to Rank 1. Rank 1 collects results from Rank 2 and Rank 3, calculates the final result and forward it to Rank 0. Now, there are two possibilities that Rank 1 either receives message from Rank 2 then Rank 3 or vice versa. Both of these message orderings are shown in Figure 6 where labeling on arrows show the order of messages. Receiving messages in different orderings produce different final averages. Results generated from both orderings at Rank 1 produces different final averages as shown in Table V. These orderings occur due to the unpredictable behavior of network. Message ordering in Figure 6a produces average 8.5 which is correct, whereas message orderings in Figure 6b produces seven, which is incorrect.

TABLE V: Message Reception Orderings at Rank 1

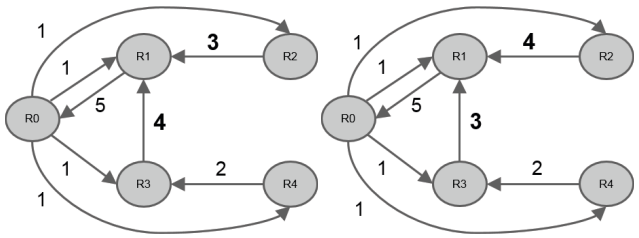
Sr. No.	Local Average	Message Ordering	Average
1.	Rank 1[2.5]	Rank 2[6.5], Rank 3[12.5]	8.5
2.	Rank 1[2.5]	Rank 3[12.5], Rank 2[6.5]	7

TABLE VI: Roles of Threads

Thread	Roles	Level
Rank 0	Initiator, Terminator	-
Rank 1	Ordinary, Intermediary	0
Rank 2	Ordinary	-
Rank 3	Ordinary	-
Rank 4	Ordinary	-

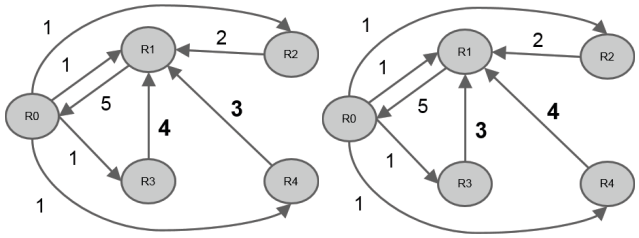
C. Running Average with Common Receiver

In this example, running average is calculated using different work distribution and collection algorithm. Fewer intermediary workers for many ordinary workers produce many message interleavings which may produce incorrect results. Roles are assigned according to Table VI. In this example, Rank 0 has the initiator and terminator role, same as previous example. Rank 2, Rank 3 and Rank 4 act as ordinary workers



(a) Correct Message Ordering (b) Incorrect Message Ordering

Fig. 6: Message Ordering with Distributed Receivers



(a) Correct Message Ordering (b) Incorrect Message Ordering

Fig. 7: Message Orderings with Common Receiver

TABLE VII: Work Distribution by Initiator (Rank 0)

Ordinary Worker	Work	Average
Rank 1	[1]	1
Rank 2	[2]	2
Rank 3	[3,4]	3.5
Rank 4	[5,6,7,8]	6.5

TABLE VIII: Message Reception Orderings at Rank 1

Sr.	Local Average	Message Ordering	Average
1.	Rank1[1]	Rank2[2], Rank3[3.5], Rank4[6.5]	4.5
2.	Rank1[1]	Rank2[2], Rank3[6.5], Rank3[3.5]	3.75
3.	Rank1[1]	Rank3[3.5], Rank2[2], Rank4[6.5]	4.3125
4.	Rank1[1]	Rank3[3.5], Rank4[6.5], Rank2[2]	3.1875
5.	Rank1[1]	Rank4[6.5], Rank2[2], Rank3[3.5]	3.1875
6.	Rank1[1]	Rank4[6.5], Rank2[6.5], Rank2[2]	2.8125

only and send their results to Rank 1. Rank 1 acts as intermediary worker and receives the result from other ordinary workers. After that, it sends results to Rank 0 terminator thread. Rank 0 distribute the workload according to Table VII i.e. workers receive different number of integers for average calculation. Due to many ordinary workers for one intermediary worker, intermediary (Rank 1) can receive messages in various orders, and two of them are shown in Figure 7. Table VIII is showing all possible orderings. Only first ordering gives the correct result in which Rank 1 receives the message from Rank 3 followed by Rank 4, as shown in Figure 7a. All other are wrong, and one of the orderings is shown in Figure 7b in which Rank 4 message is received followed by Rank 3 message produce incorrect results.

Sample output of running average example with input of first sixteen integers is shown in Figure 8. There are nine threads (MPJ processes) which produce ninety-six interleavings for complete state space exploration, excluding states pruned by JPF using POR. Only one interleaving at line 6 is producing correct output (8.5) whereas all other interleavings produce incorrect average.

```
1  Executing command: java -jar C:\Users\SOHAIB\workspace\jpf-core\build\RunJPF.jar +shell.port=4242
       C:\Users\SOHAIB\workspace\MPI_Verifier\src\MPI_Verifier.jpf
2  JavaPathfinder core system v8.0 - (C) 2005-2014 United States Government. All rights reserved.
3  ============================================================ system under test
4  MPJ_Express_Model.main()
5  ============================================================ search started: x/xx/17 2:07 AM
6  Final Average:8.5
7  Final Average:8.125
8  Final Average:7.84375
9  Final Average:7.65625
10 Final Average:8.40625
11 Final Average:7.0
12 ...
98 Final Average:3.5195312
99 Final Average:4.4453125
100 Final Average:5.0195312
101 Final Average:3.8945312
102 Final Average:3.5195312
103 ============================================================ results
104 no errors detected
105 ============================================================ statistics
106 elapsed time:       00:00:19
107 states:             new=5825,visited=4448,backtracked=10273,end=96
108 search:             maxDepth=21,constraints=0
109 choice generators:  thread=5824 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=5823), data=0
110 heap:               new=137228,released=222300,maxLive=1182,gcCycles=10273
111 instructions:       26479836
112 max memory:         603MB
113 loaded code:        classes=84,methods=1681
114 ============================================================ search finished: x/xx/17 2:07 AM
```

Fig. 8: Sample Output of Running Average Example

TABLE IX: Execution Time Taken by Our MPJ Model and MPJ Express for Running Average Example

No. of Threads (1 Sender, n Receivers)	Time Taken by	
	MPJ Model	MPJ Express
1(S), 2(R)	100 ms	237 ms
1(S), 4(R)	1,000 ms	342 ms
1(S), 6(R)	3,000 ms	465 ms
1(S), 8(R)	9,000 ms	497 ms
1(S), 10(R)	109,000 ms	570 ms
1(S), 12(R)	2,896,000 ms	683 ms

D. Scalability of our Approach

We have compared execution time of running average example using our MPJ model with execution time taken by MPJ Express. An Intel Pentium 4 machine with 4GB of RAM was used for this experiment. Our evaluation shows that our MPJ Express Model successfully explored all possible interleaving of MPJ program in the finite amount of time, as shown in Table IX. Scalability of our approach is affected by the number of potential receivers for a massage and not by the total number of processes or by direct point-to-point communication, hence our examples vary the number of receivers receiving a message to show effect on scalability. Table IX shows different execution times for program with single sender/distributor and n receivers e.g. for first row, 1(S),2(R) means 1 sender and 2 receivers. Second column shows the time it takes to run the program using our MPI Java model with all possible interleavings. Whereas, third column shows the time it takes to run one execution of the program on MPJ Express framework which might be right/wrong due to non-deterministic nature of the program. These times shows how scalability is affected by the number of receivers willing to receive a message. Partial Order Reduction (POR) mechanism of JPF helps to avoid state explosion problem in our model.

We have chosen order-sensitive applications which exhibit a high degree of non-determinism for our evaluation. MPI applications using a lot of point-to-point communication i.e. sending to a specific receiver do not generate multiple possible orders. Only messages that can be received by one of many receivers require model checking. As shown above, in Figure 8, ninety-six interleavings are produced with nine threads, excluding pruned interleavings due to POR, for calculation of running average. Most real world applications do not have so many non-deterministic paths which is one of the obvious reasons to choose small examples to evaluate our work. Therefore, our approach takes more time as the number of threads increases due to increase in all possible interleavings to be explored, which can be seen through the comparison of time taken by MPJ Express with the time taken by our model. For the real-world applications, this performance gap is much smaller due to less non-determinism, which is not of much interest for evaluation of our approach to verify MPI Java programs.

VI. Related Work

Efforts to find bugs in MPI-based programs can be broadly categorized into two main categories: testing and debugging

methods, verification methods.

Debugging and testing tools work on concrete inputs which might miss some of the concurrency bugs. DDT[2] and TotalView[3] work best when bugs can be replayed, but it is difficult for MPI programs due to their non-deterministic nature. Our work does not depend upon the reply behavior of the programs. Marmot [17], Umpire [27], MUST [15], the Intel Trace Analyzer and Collector [20] record the runtime information by intercepting MPI Calls but these tools are scheduling and input dependent. Our work tries to explore all the interleaving by model checking MPJ program without need of intercepting any MPI calls. ValiMPI [13] provides support for specifying testing criterion for MPI programs to obtain better quality and coverage of generated test cases. It works for parallel programs written in C language and extends Instrumentation Description Language (IDeL) to deal with such programs. In contrast to ValiMPI, our work runs on the same unchanged program without using any additional information or Instrumentation Description Language. Hursey et al. [16] presented MPI Testing Tool (MTT) for regression testing of MPI Programs across organizations and environments. OpenMPI project uses it for regression testing which means it is also meant for C/C++ MPI programs. This work is meant for regression testing of MPI programs as opposed to our work to find all the interleaving of different message ordering.

DAMPI [29] and ISP [25] check for the non-deterministic behavior of MPI programs by re-executing the program when wildcard receive is encountered. So, these tools work only when non-deterministic choice is made and also depend on the input of the MPI program. Our work is input independent and does not require re-execution of program on non-deterministic choice. Parallel Control Flow Graph (pCFG) [4] approach perform the static analysis on the activities which is hard to automate. Our proposed solution is based on model checking of programs and free from the problems faced by static analysis technique. MPI-SPIN [23] model checks the MPI programs, but it requires user built abstract model of the program using MPI-SPIN language which becomes very complex for large programs. Our proposed model does not require any abstract model of the program and does not require learning of any new language. MPISE [7] perform the symbolic execution of the C MPI Program and employs on-the-fly scheduling algorithm to reduce the interleaving exploration but this work is limited to C language and requires specification of certain inputs and symbolic variables. In our proposed work, there is no need of specification of symbolic variable or symbolic input. Our model can explore all the possible interleavings of the program using model checking of MPJ program. Some other works tried to verify specific properties of MPI programs which can only be used in that specific scenario. Siegel et al. [24] presented a method for verification of MPI programs involving floating point and other complex numerical computations. It performs model checking along with symbolic execution on the program and check for the equivalence of sequential version of the same program with the parallel version. This work does not involve different message ordering exploration as opposed to our proposed work. Gopalakrishnan et al. [10] presented the set of tools that specialize formal methods for the verification of real-world MPI applications. These tools broadly based on debugging techniques, symbolic analysis, static and dynamic analysis. Again, these tools lack the capability to verify the message ordering in MPI Java program without using any additional information.

Some other tools also exist for specific language and specific error checking. MPI-CHECK [19] replace MPI calls with MPI-CHECK's own version of same calls, and it only works for Fortran 90 programs. It is focused on static issues in correct MPI usage unlike dynamic message ordering bugs. MPIDD [12] and MPIRace-Check [21] only focuses on deadlock detection and race detection in MPI programs respectively. As compared to MPIDD and MPIRace-Check, our work tries to explore message orderings in MPJ programs.

Leungwattanakit et al. [18] presented modular model checking technique in which model checker (JPF) only verifies a single process in a distributed system and interact with other processes running normally outside verification tool. The interaction between system under test (SUT) and other peer processes is cached for state backtracking by JPF along with checkpointing tool to capture peer state executing outside model checker. This technique limited to verification of SUT and can not handle non-determinism in peers. It also misses some faults due to unavailability of complete state space of the system and only works for applications using client-server architecture. As opposed to work mentioned above, our work performs verification of complete system without producing any additional false negative. NetStub [3] replaces networking API of Java with manually created stub classes to capture interaction of single process with distributed system. It requires model of the environment for verification which is not required by our work. Above work is also limited to modeling of network library and does not cater for non-deterministic nature of distributed systems. CADP [8] is a toolbox for design of concurrent systems and allow to model the interaction between MPI and distributed cache coherence protocol but it does not offer any tool for verification unchanged of MPI Java program.

Galen et al. [26] presented a tool, Maggy, for translating MPJ-based Java programs into Promela language for verification of safety properties. Whereas, our work does not require any additional language. Same MPJ program can be verified for different message orderings using our work without need of learning any new language, for example, Promela specifications. This work only supports model checking based on Promela specifications and requires manual editing of Promela files. Also, it only supports logical control flows present in the main function and static target of MPJ send/receive functions. Whereas, our work is not limited to only main function of the

[2]Allinea DDT: http://www.allinea.com/products/ddt

[3]TotalView: http://www.roguewave.com/products-services/totalview

program and does not require specific format of send/receive function. Any MPJ program using point-to-point send/receive can be model checked using our proposed work even if main of program is calling some other functions. Our work can verify any MPJ program containing point-to-point communication without knowledge of any other language and constraints on logical flow of the program.

VII. Conclusions

Distributed nature of MPI programs makes it difficult to model check MPI Programs and very little effort has been made to verify MPI Java Programs. In this paper, we have shown that using the Java PathFinder (JPF) explicit state software model checker; we can model check MPI Java programs. We developed a Java based model of MPJ Express, where processes are modeled as threads, and which can run unmodified MPI Java programs on a single system. This model enabled us to adapt the JPF model checker to verify our model running real MPI Java programs. We evaluated our approach using small examples where model checking revealed message orders that would result in incorrect system behavior. JPF was extended by a listener, which creates choice points and controls the scheduling. Further, we have extended JPF by introducing a package which helps us in logging the states and creating relationship between them.

Developing concurrent and distributed software requires the programmer to be aware of non-determinism, data races, and deadlocks. While distributed software enables scalability, correctness and verification become a serious issue. We believe, model checking of unmodified distributed systems is a promising technique to enable development of more reliable distributed systems.

References

[1] M. Baker and B. Carpenter. MPJ: A Proposed Java Message Passing API and Environment for High Performance Computing. In *Parallel and distributed processing*, pages 552–559. 2000.

[2] M. Baker, B. Carpenter, and A. Shafi. MPJ Express: Towards Thread Safe Java HPC. In *IEEE International Conference on Cluster Computing*, pages 1–10. IEEE, 2006.

[3] E. Barlas and T. Bultan. Netstub: A Framework for Verification of Distributed Java Applications. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 24–33. ACM, 2007.

[4] G. Bronevetsky. Communication-Sensitive Static Dataflow for Parallel Message Passing Applications. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–12, 2009.

[5] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model Checking: Algorithmic Verification and Debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[7] X. Fu, Z. Chen, Y. Zhang, C. Huang, and J. Wang. MPISE: Symbolic Execution of MPI Programs. In *16th International Symposium on High Assurance Systems Engineering (HASE)*, pages 181–188, 2015.

[8] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox For The Construction And Analysis of Distributed Processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013.

[9] P. Godefroid, J. Van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032. Springer Heidelberg, 1996.

[10] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky. Formal Analysis of MPI-based Parallel Programs. *Communications of the ACM*, 54(12):82–91, 2011.

[11] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT press, Cambridge, MA, USA, 2nd edition, 1999.

[12] W. Haque. Concurrent Deadlock Detection in Parallel Programs. *International Journal of Computers and Applications*, 28(1):19–25, 2006.

[13] A. C. Hausen, S. R. Vergilio, S. R. Souza, P. S. Souza, and A. S. Simao. A Tool for Structural Testing of MPI Programs. *8th IEEE Latin-American Test Symposium, LATW (March 2007)*, 2007.

[14] K. Havelund and T. Pressburger. Model Checking Java Programs Using Java Pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[15] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 30, 2012.

[16] J. Hursey, E. Mallove, J. M. Squyres, and A. Lumsdaine. An Extensible Framework for Distributed Testing of MPI Implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 64–72. Springer, 2007.

[17] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch. MARMOT: An MPI Analysis and Checking Tool. *Advances in Parallel Computing*, 13:493–500, 2004.

[18] W. Leungwattanakit, C. Artho, M. Hagiya, Y. Tanabe, M. Yamamoto, and K. Takahashi. Modular Software Model Checking for Distributed Systems. *IEEE Transactions on Software Engineering*, 40(5):483–501, 2014.

[19] G. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou. MPI-CHECK: A Tool for Checking Fortran 90 MPI Programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[20] P. Ohly and W. Krotz-Vogel. Automated MPI Correctness Checking What if there was a magic option? In *Proceedings of the 8th LCI International Conference on High-Performance Clustered Computing*, pages 19–25, 2007.

[21] M.-Y. Park, S. J. Shim, Y.-K. Jun, and H.-R. Park. MPIRace-Check: Detection of Message Races in MPI Programs. In *Advances in Grid and Pervasive Computing*, pages 322–333. Springer, 2007.

[22] W. U. Rehman, M. S. Ayub, and J. H. Siddiqui. Verification of MPI Java Programs using Software Model Checking. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 55–56. ACM, 2016.

[23] S. F. Siegel and G. S. Avrunin. Verification of MPI-based Software for Scientific Computation. In *Model Checking Software*, pages 286–303. 2004.

[24] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Combining Symbolic Execution With Model Checking To Verify Parallel Numerical Programs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):10, 2008.

[25] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 285–286, 2008.

[26] R. van Galen. Towards Verification of MPJ-based Java Programs. *15th Twente Student Conference on IT*, 15, 2011.

[27] J. S. Vetter and B. R. de Supinski. Dynamic Software Testing of MPI Applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 51–51, 2000.

[28] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[29] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. R. De Supinski, M. Schulz, and G. Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2010.