FORMAL ANALYSIS FOR MPI-BASED HIGH PERFORMANCE COMPUTING SOFTWARE

by

Robert Lee Palmer

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

School of Computing
The University of Utah

August 2007
SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Robert Lee Palmer

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair:    Ganesh Gopalakrishnan

Robert M. Kirby

Gary Lindstrom

Sriram K. Rajamani

Konrad Slind
To the Graduate Council of the University of Utah:

I have read the dissertation of Robert Lee Palmer in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Ganesh Gopalakrishnan
Chair, Supervisory Committee

Martin Berzins
Chair/Dean

David Chapman
Dean of The Graduate School
ABSTRACT

Model checking has been applied to concurrent system paradigms such as cache coherence protocols, device drivers, and threads based programming with great success in finding errors that are otherwise difficult to diagnose and remove. The primary reasons for this success are: (i) the set of communication idioms, used in a given paradigm, is relatively small and well understood by the model checking tools, (ii) model checking examines all interesting execution traces in an automatic and systematic way, and (iii) the properties being checked are amenable to model checking analysis.

In this dissertation, the application of model checking analysis to MPI-based parallel programs, which have a rich set of communication idioms available in the form of a communication library, is examined. To facilitate model checking analysis, formal model of the MPI 1.1 standard is proposed using the Temporal Logic of Actions (TLA+). This model serves as an executable reference for answering questions about how the various operations may interact and describes the communication idioms using a formal logic notation. Using the model, one can pose pithy litmus tests with respect to the MPI standard that are analyzed using TLC, a model checker for the TLA+ language.

A representative subset of communication primitives are chosen (one send, receive, wait, and test operation, along with barrier) and the independence characteristics of these operations are shown. Leveraging this independence information, this dissertation presents a dynamic partial-order reduction algorithm to facilitate scalability in model checker based reasoning. The partial-order reduction algorithm and this subset of communication primitives has been implemented in a prototype model checker called MPIC.
To facilitate litmus test based examination of MPI-based C programs, a model extraction framework is presented. This framework is integrated with the Microsoft Visual Studio development environment, combining TLA+ program models with the TLA+ MPI model. Initial results are encouraging as intricate scenarios for a small number of process can be examined. The framework also creates MPIC models, allowing for the examination of somewhat larger programs in a limited setting.
To my dear wife Laurie.
CONTENTS

ABSTRACT ................................................................. iv
LIST OF FIGURES ....................................................... xi
LIST OF TABLES ........................................................ xiii
ACKNOWLEDGMENTS .................................................. xiv

CHAPTERS

1. INTRODUCTION ...................................................... 1
   1.1 Problems Addressed by this Dissertation ..................... 5
   1.2 Primary Contributions of this Dissertation ............... 6
   1.3 Organization of the Dissertation ........................... 11

2. A SEMANTIC MODEL OF MPI ................................. 12
   2.1 Introduction to TLA+ ........................................ 13
   2.2 Introduction to MPI ......................................... 16
      2.2.1 A Description of the MPI Primitive Naming Convention .. 16
      2.2.2 A Simple Example of an MPI Program ................. 17
   2.3 Overview of MPI Model Architecture ..................... 19
   2.4 Modeling Approach ........................................ 20
      2.4.1 Data Structures in the Model ...................... 22
         2.4.1.1 Communicators ................................ 22
         2.4.1.2 Groups ........................................ 22
         2.4.1.3 Collective Context ............................ 23
         2.4.1.4 Requests ...................................... 23
         2.4.1.5 Messages ..................................... 24
         2.4.1.6 Statuses ...................................... 24
   2.5 Invariants and Assertions Included in the TLA+ MPI Model . 25
   2.6 Abstractions and Simplifications Imposed on the TLA+ MPI Model 27
   2.7 Model Granularity Issues ................................ 28
      2.7.1 Matching and Pairing Messages .................... 29
      2.7.2 Transmitting Messages ............................. 29
      2.7.3 Buffering Messages ................................. 31
      2.7.4 Canceling a Message ............................... 31
      2.7.5 Full Transition Relation ........................... 32
   2.8 Initiating Point to Point Messages .................... 32
      2.8.1 Immediate Mode Operations ........................ 33
2.8.2 Persistent Mode Operations .................................................. 33
2.9 Completing Point to Point Messages ........................................ 33
  2.9.1 Waiting for Completion .................................................... 33
  2.9.2 Testing for Completion .................................................... 36
  2.9.3 Variants: All, Any, and Some .......................................... 37
2.10 Compositional Modeling of Standard Mode Operations ............... 37
  2.10.1 Models of the MPI_Ssend, MPI_Bsend, MPI_Ssend, MPI_Rsend,
        and MPI_Recv Primitives .............................................. 38
  2.10.2 A Model of the MPI_Sendrecv Primitive .............................. 39
  2.10.3 A Model of the MPI_Sendrecv_replace Primitive .................... 40
2.11 Collective Operations Included in the Model .......................... 41
2.12 The Initial State of the Model ............................................ 42
2.13 Issues Raised by Modeling ................................................ 42
2.14 Related Work ...................................................................... 43
2.15 Summary ............................................................................ 44

3. DYNAMIC PARTIAL-ORDER REDUCTION FOR MPI ........... 46
  3.1 Motivating Examples .......................................................... 48
  3.2 Language Execution and Communication Semantics ..................... 50
    3.2.1 Assumptions and Properties of Interest ............................. 59
    3.2.2 Independence Characteristics of MPI Operations .................. 59
    3.2.3 Proof of the Independence Theorems for MPI Operations ...... 61
  3.3 Dynamic Partial-order Reduction for MPI ............................... 65
    3.3.1 Proof of Correctness of the Reduction Algorithm ............... 69
  3.4 Experiments Using DPOR for MPI ....................................... 76
  3.5 A Detailed Example .......................................................... 77
  3.6 Related Work ................................................................... 78
  3.7 Summary ........................................................................... 84

4. MODEL EXTRACTION AND SIMPLIFICATION ..................... 85
  4.1 An Overview of the Modeling Framework ................................. 86
    4.1.1 Program Simplifications ............................................. 87
    4.1.2 Program Modeling ..................................................... 88
    4.1.3 Error Trail Generation ............................................... 90
    4.1.4 Examples .............................................................. 90
  4.2 The MPIC Intermediate Representation ..................................... 91
    4.2.1 Program States ....................................................... 92
    4.2.2 Program Transitions ................................................ 92
      4.2.2.1 The EnterFunctionTransition Transition ....................... 92
      4.2.2.2 The ReturnTransition Transition ............................... 93
      4.2.2.3 The CallTransition Transition .................................. 93
      4.2.2.4 The BranchTransition Transition .............................. 93
      4.2.2.5 The ValueTransition Transition ............................... 93
      4.2.2.6 The CompareTransition Transition ............................ 94
      4.2.2.7 The SwitchTransition Transition .............................. 94
    4.2.3 Transition Operands ................................................ 94
6.3.1.2 Using the Model ................................................. 119
6.3.2 Future Directions for the Model Extraction Framework .... 119
  6.3.2.1 Interpolant-based Model Checking ......................... 119
  6.3.2.2 Language Feature Extraction ................................ 119
  6.3.2.3 Model Feature Extraction ................................... 120
  6.3.2.4 Abstraction Refinement ..................................... 120
6.3.3 Future Directions for Model Checking MPI Programs ....... 120
  6.3.3.1 Independence of MPI Operations ......................... 120
  6.3.3.2 Implementing Additional MPI Operations ................. 120
  6.3.3.3 Symmetry for MPI ........................................... 121

APPENDICES

A. THE TLA+ MODEL OF THE MPI 1.1 STANDARD .............. 122
B. AN EXAMPLE MPI PROGRAM MODELED IN TLA+ ............. 174
C. THE TOP LEVEL TLA+ SPECIFICATION ....................... 180
REFERENCES ............................................................... 183
LIST OF FIGURES

1.1 A Program Pseudo-code Fragment for two MPI processes executing in parallel. ......................................................... 4
1.2 The C-based Litmus Test used to answer Question 4 on Page 4. ...... 10
2.1 A TLA+ specification of the classic Dining Philosophers problem. .... 14
2.2 A simple “Hello World” example C program using MPI. .............. 18
2.3 The architecture of the TLA+ MPI model. ............................... 19
2.4 The Transmit rule for moving data between processes in the TLA+ MPI model. ............................................................ 30
2.5 The top level next-state relation for the TLA+ MPI model. .......... 32
2.6 The TLA+ model of MPI_Wait, part 1. ............................... 34
2.7 The TLA+ model of MPI_Wait, part 2 .................................. 35
2.8 A template for transforming MPI_Send into an MPI_Isend followed by a MPI_Wait. .............................................. 38
2.9 A template for transforming MPI_Sendrecv into an MPI_Isend and an MPI_Irecv transition followed by a pair of MPI_Wait transitions. ................................. 39
2.10 A template for transforming MPI_Sendrecv_replace into an MPI_Isend and an MPI_Irecv transition followed by a pair of MPI_Wait transitions and an assignment. ........................................ 40
2.11 The state machine for the barrier synchronization protocol. ........ 41
3.1 A program pseudo-code that contains a deadlock caused by a mismatched handshake. ........................................ 49
3.2 A program pseudo-code that contains an assertion violation caused by nondeterministic message order. ...................... 50
3.3 A program pseudo-code fragment illustrating the role of dynamic information in partial-order reduction for MPI. ............... 51
3.4 The Grammar of the MPIC modeling language. ....................... 52
3.5 Helper functions used in the communication semantics. ............. 54
3.6 The semantic definitions of Assignment, Assert, Goto, Issend, and Irecv. ............................................. 55
3.7 The semantic definitions of Barrier_init and Barrier_wait. .......... 56
3.8 The semantic definition of Wait. ........................................... 57
3.9 The semantic definition of Test. ................................. 58
3.10 Definition of the predicate Complete.......................... 61
3.11 Dynamic partial-order reduction based stateful depth-first search. ... 67
3.12 A dynamic partial-order reduction based stateless depth first search model checking algorithm. ................................. 68
3.13 Three processes executing concurrently. ....................... 71
3.14 A program pseudo-code fragment containing a non-deterministic receive operation. ........................................ 78
3.15 Part of the state space visited by the model checker when examining the program of Figure 3.14 using the algorithm of Figure 3.12. ....... 79
4.1 The architecture of the MPIC verification environment. .......... 87
4.2 A TLA+ rule for memory allocation used with MPI program models. 89
4.3 A Transition template for TLA+ program models. ............... 89
4.4 An algorithm for reverse depth-first program graph traversal. ...... 96
4.5 An algorithm for chaotic iteration based program graph traversal. ... 97
5.1 An algorithm for stateless depth first search based model checking. . . 111
LIST OF TABLES

2.1 The point-to-point operations included in the TLA+ specification.

2.2 The additional MPI operations modeled to enable model checking based reasoning on MPI-based parallel programs.

4.1 The number of states generated and execution time in seconds on some simple examples.
ACKNOWLEDGMENTS

There are many people who have helped me in the long road to the completion of my Ph.D., and to whom I owe many thanks. Here I name just a few. My advisor and supervisory committee at The University of Utah: Ganesh who introduced me to formal verification and model checking, Konrad who taught me theorem proving, and Gary and Mike who were there along the way to lend support and insights. I should also thank Sriram Rajamani, my external examiner, for bringing me to Microsoft Research to work on Zing and take in the MSR experience which ultimately taught me why I want to do research for a living.

I gratefully acknowledge the help, support, consolation, and constant encouragement of my wife and sons: Laurie, Andrew, and James, without whom none of this would have happened.

Additional thanks goes to the Microsoft HPC guys, in particular Kyril Faenov, Dennis Crain, and Shahrokh Mortazavi for supporting this line of research and suggesting that we use the Phoenix compiler framework as a front end. In addition to the above I should thank John O’Leary, Tom Schubert, and Roope Kaivola who let me spend a great six months at Intel doing floating point verification. Thanks also to Vlad Levin my mentor while at MSR and Shaz Qadeer for all the hard work trying to get the transaction-based reduction proofs correct.

I would also like to thank the undergrads who have worked with me and helped in various aspects of my research through the years: Michael Delisi, Geof Sawaya, Sonjong Hwang, Eun Yong Kang, Jacob Tripp, and Omid Sadaati. I have learned a lot from each of you. I should mention also my appreciation for the other graduates in our group, both present and past, who have had an effect on my work, the office staff and academic advisors who have always been cheerful and ready to help, and extended family, friends, and neighbors who have always given their support.
CHAPTER 1
INTRODUCTION

To continue leveraging the increase in transistor density, the software engineering community must adopt some form of concurrency and concurrent programming paradigm. However, writing a correct concurrent program is difficult. Even when "correctness" disregards the calculations of the program and considers only reactive issues such as the absence of deadlock, getting the program to be correct is difficult.

One reason for the difficulty of concurrent programming is the unexpected ways that the various individual processes can interact. This is largely due to the exponential number of serializations of the processes’ execution interleaving orders, many of which are not, and perhaps cannot be, anticipated by the application developer.

A second problem arises from the communication primitives. The semantics of the individual operations are often incompletely defined and in some instances have conflicting definitions between platforms; certainly no formal definition exists. The result of this ambiguity is confusion about, and abuse of the semantics of the communication operations in various use scenarios.

Formal verification methods have been playing an increasingly important role in the area of reasoning about concurrent programs because they address precisely these two issues, namely (i) systematic exploration of all interesting execution interleaving orders, and (ii) precise formalization of communication and execution semantics. Yet, formal verification research with respect to concurrent programming is typically focused on simplified models of message passing via abstract channels or shared variable communications using idealized locks or semaphores that do not fully reflect the semantics of real systems.
In practice, concurrent programs use libraries, usually in concert with some language features, to expose the concurrency primitives of the underlying platform to applications developers. Java threads, POSIX threads for C/C++, message passing via MPI, and others have an application programming interface (API) that either implies some semantic notions or explicitly enforces them. What is lacking for these systems are formal verification methodologies and tools that take library semantics into account and allow for reasoning about the individual programs with respect to the libraries that are used.

Formal verification is needed to advance the reliability of parallel programs. As the explicit parallelism used to express programs increases, those rare errors, such as deadlocks and data races, that are difficult to reproduce in testing will be experienced every day by many users. Without sound methods to insure that concurrency related defects are not included with software released to market, the benefits of increased transistor density, and the resulting increase in performance from greater parallelism will be nullified by the repeated embarrassment of unresponsive and unpredictable software. While software engineers may be sympathetic, it is not clear that the general purchasing public will be as kind.

Consider the arena of High Performance Computing (HPC). For many years the HPC community has exploited parallelism to tackle the most demanding of computational problems. In HPC, library based communication is the dominant communication paradigm for distributed applications. This is primarily due to the popularity of libraries that implement the Message Passing Interface (MPI) standard [82]. The Message Passing Interface has become a de facto standard in HPC, and is being actively developed and supported through several implementations [25, 32, 56, 81] designed to run on a plethora of architectural platforms.

The MPI standard offers a rich set of communication primitives that allow the user to choose the subset right for the given application. In instances where overlapping communication with computation can be done efficiently, MPI offers constructs to facilitate. Instances that require a lock step approach can again be facilitated. Moreover, the 2.0 version of the standard [83] describes one-sided
operations that allow processes to open so-called “memory windows” to other processes in the group.

MPI is a standard for overall behavior and portability. Therefore, MPI programs are often manually or automatically (e.g., [17]) re-tuned when ported to another hardware platform, for example by changing its basic primitives (e.g., `MPI_Send`) to specialized versions (e.g., `MPI_Isend`). The fact that MPI-1 supports over 128 primitives and MPI-2 supports over 300 is largely to facilitate such transformations.\(^1\) While MPI programs avoid the pervasive global state that makes shared memory thread programs highly error prone, they are still extremely error-prone, as numerous studies show (e.g., [18]). In this context, it is crucial that the developers of MPI programs, and particularly those performing code tuning are aware of the fine details of MPI semantics. Unfortunately, such details are far from obvious.

For illustration, consider the MPI pseudo-code involving two processes shown in Figure 1.1. Process P1 is designed to issue two immediate mode sends (the first being a synchronous send) to process P0 (hence the name `MPI_Issend`, `I` indicating immediate mode and `s` indicating synchronous), while Process P0 is designed to post two immediate-mode receives. Consider some simple questions pertaining to the execution of this program:

1. **Is it guaranteed that rcvbuf1 will eventually contain the message sent out of sendbuf1?** The answer is ‘yes,’ since MPI guarantees in-order message delivery.

2. **When can the buffers be accessed?** Since all sends and receives use the immediate mode, the handles that these calls return have to be tested for completion using an explicit `MPI_Test` or `MPI_Wait` (suppressed for brevity in the pseudo-code) before the associated buffers are allowed to be accessed (written or even read).

---

\(^1\)It is widely known that MPI programs typically use only about a dozen or so of the 300 MPI library calls — but the precise dozen chosen depends on the applications being programmed, as well as the hardware platform on which the program runs.
1 P0: if(rank==0){
2       MPI_Irecv(rcvbuf1, from 1);
3       MPI_Irecv(rcvbuf2, from 1);
4 ...
5 }
6 P1: if(rank==1){
7       sendbuf1=6;
8       sendbuf2=7;
9       MPI_Isend(sendbuf1, to 0);
10      MPI_Isend(sendbuf2, to 0);
11 ...
12 }

Figure 1.1. A Program Pseudo-code Fragment for two MPI processes executing in parallel.

3. **Will the first receive always complete before the second?** No such guarantee exists (the second may complete first), as these are *immediate mode* receives which are guaranteed only to be *initiated* in program order.

4. **What is guaranteed about the matching receive when the first send completes?** It is guaranteed that this receive has been *posted*. This is because the first send is a *synchronous* send, which forces a rendezvous with the posting of the matching receive.

The MPI 1.1 reference standard [82] is an informal, non machine-readable document that offers natural language descriptions of the individual behaviors of MPI primitives. It does not support answering the above kinds of simple questions in any tractable and reliable way. Moreover, running test programs, using actual MPI libraries, to reveal answers to the above kinds of questions is also futile, given that (i) various MPI implementations may specialize the semantics in a variety of ways, and (ii) the execution interleaving order of a given test run may mask the answer.

Distributed programming with MPI provides a good application context for formal verification. Many techniques proposed in the formal verification community can be applied to answer specific problems such as those posed above. To
enable applications of these techniques requires the creation of a domain specific formal verification methodology that benefits both formal verification and high performance computing communities.

There should first be a formal definition of the communication semantics of MPI. Once the semantics of the communication operations have been formally defined, formal methods in the form of model checking based reasoning can be applied to MPI-based parallel programs — stating properties of models of MPI-based parallel programs and using model checking [15] to prove these properties — effectively answering questions such as those posed above. Model checking has been applied by formal verification experts to MPI programs, finding deep-seated errors [72, 76].

Explicit state model checking techniques are limited by available memory. This is due in part to the complexity of computing the next state relation, and in part to the potentially exponential number of states that can be created under an interleaving semantics. The exponential number of reachable states, often known as state explosion requires attention in this work because of the large numbers of processes that are often involved in MPI programs.

Several techniques have been proposed to help mitigate the state explosion problem. The partial-order reduction technique [15, Chapter 10] works to reduce the effects of state explosion by safely delaying the execution of some processes while allowing others to run. In this way many of the redundant execution interleaving orders are ignored by the model checker while still visiting enough of the reachable state space of the model to prove the desired properties. Partial-order reduction is a natural approach to containing state explosion in this setting because MPI programs interact only when the MPI functions — such as various forms of send, receive, or barrier synchronization — are invoked.

1.1 Problems Addressed by this Dissertation

This dissertation applies formal verification methods to the domain of High Performance Computing. In HPC there are some dominant paradigms: Single Program Multiple Data (SPMD) programming style, and the use of explicit message
passing between concurrent processes via libraries implementing the MPI standard. This work seeks to leverage these paradigms in the application of formal methods to high performance computing applications.

There is a need for formalization with respect to the library semantics. Without such, any attempts at formal analysis are simply debugging aids and cannot make any claims concerning correctness. This formalization should be readable by experts in MPI that are not fluent in the formal methods vernacular. Moreover it should be amenable to use as a communication subsystem or library in a model checking framework.

The second problem that is immediate with respect to the concurrency present in HPC applications is containing the state explosion problem when applying model checking analysis. To apply a partial-order reduction, the independence of the transition semantics must be made known. Independence is a relation on transitions that tells us when executing the two transitions in either order will result in the same global state.

A third problem this dissertation addresses is automatic program modeling for SPMD style MPI-based C programs. The process of modeling programs is often error prone and laborious. In addition, it requires some formal methods expertise on the part of the person doing the modeling — a difficult barrier when applying formal methods to a new domain.

1.2 Primary Contributions of this Dissertation

The primary contributions of this dissertation are the following: In the area of applied formal verification — (i) A formal, executable, semantic model of the point to point communication operations included in the MPI standard. This model facilitates deep understanding of the interactions possible between MPI constructs by the application of litmus tests. (ii) A model checker prototype that internalizes a subset of the formal semantics and leverages them to effect partial-order reduction. (iii) A model extraction and error trail simulation framework to generate and simplify program models from C source code as well as simulate error trails in the
setting of the Microsoft Visual Studio parallel debugger environment. In the area of theoretical formal verification — (i) A new algorithm for domain specific dynamic partial-order reduction of MPI-based parallel program models. (ii) A number of theorems, and proofs of correctness of those theorems, regarding independence of MPI operations under the proposed model. (iii) A proof that the algorithm in (i) preserves local assertions, deadlocks, and the presence of cycles.

With respect to the verification framework, the following features have been developed:

1. The framework permits designers to explore the MPI semantics in the setting of MPI programs written in C, by automatically extracting a TLA+ model of the program, embedding the MPI calls as TLA+ operators in the program model, and linking it to the TLA+ model of MPI functions. The exploration happens through model checking, and not through concrete executions. Error traces produced by the model checker are, however, displayed in a user-friendly way by driving the Microsoft Visual Studio debugger to execute the original program code following the error trace.

2. The framework incorporates two model checkers: TLC, a model checker that works directly off the formal semantic definitions in TLA+ [91]; and MPIC [68], a model checker that encodes the communication semantics of MPI directly as C# program code.

3. The communication semantics of a small representative subset of MPI is incorporated into MPIC by faithfully following the TLA+ definitions. In addition, MPIC implements a dynamic partial-order reduction algorithm (DPOR) (adapted from [24]) for efficient state-space traversal. The DPOR algorithm avoids commuting independent actions, where the notion of independence was stated and manually proved using the MPI formal semantics.

The questions raised on Page 3 can be answered by writing an MPI-based C program and analyzing this program using the framework. The four questions can be answered, as follows:
1. **Question:** Is it guaranteed that \texttt{rcvbuf1} will eventually contain the message sent out of \texttt{sendbuf1}? The answer is ‘yes,’ since MPI guarantees in-order message delivery.

   **Answer:** To observe this behavior with a litmus test, assert that the data read by process 0 is: \texttt{rcvbuf1 == 6} and \texttt{rcvbuf2 == 7}. If it is possible under the semantics for other values to be assigned to these two variables, then the model checker will find the violation.

2. **Question:** When can the buffers be accessed? Since all sends and receives use the immediate mode, the handles that these calls return have to be tested for completion using an explicit \texttt{MPI_Test} or \texttt{MPI_Wait} (suppressed for brevity in the pseudo-code) before the associated buffers are allowed to be accessed (written or even read).

   **Answer:** To observe this behavior with a litmus test, move the assertions from the answer to Question 1 to any other point before the corresponding \texttt{MPI_Wait} operations. The model checker then finds violations — meaning that the data cannot be accessed on the receiver until after the corresponding \texttt{MPI_Wait}. If one adds an assignment to the variable being transmitted, after the \texttt{MPI_Issend}, yet before the corresponding \texttt{MPI_Wait} or \texttt{MPI_TEST}, the model checker discovers the assertion violation as the wrong value is passed to the receiver.

3. **Question:** Will the first receive always complete before the second? No such guarantee exists (the second may complete first), as these are immediate mode receives which are guaranteed only to be initiated in program order.

   **Answer:** To observe this behavior with a litmus test, reverse the order of the corresponding \texttt{MPI_Wait} commands for the shown \texttt{MPI_Recv} operations. If the model checker does not find a deadlock then it is possible for the operations to complete in either order.
4. **Question:** What is guaranteed about the matching receive when the first send completes? It is guaranteed that this receive has been *posted*. This is because the first send is a *synchronous* send, which forces a rendezvous with the posting of the matching receive.

**Answer:** To observe this behavior with a litmus test, employ the program in Figure 1.2. The MPI semantics for immediate mode ready send requires the corresponding receive to be posted before the `MPI_Irsend`. In the program, the tag of the messages force the second `MPI_Irecv` to match the `MPI_Issend`. The program then executes the `MPI_Wait` corresponding to the `MPI_Issend` and then post two `MPI_Irsend` operations. Now observe that the TLC model checker (in breadth-first search mode) finds that the first `MPI_Irsend` posts without error, but the second `MPI_Irsend` violates one of the assertions placed regarding the semantics of `MPI_Irsend`. Thus when the `MPI_Wait` that corresponds to the `MPI_Issend` of process 1 returns, process 0 is guaranteed to have executed the second `MPI_Irecv`, but is not guaranteed to have executed any further.

In the act of writing the formal specification of MPI, some cases of under-specifications in the natural language standard were identified (confirmed by experts [34]). While these omissions were found largely by luck, the *opposite* problem — namely, the specification itself may not correctly implement the intent of the MPI natural language standard writers needs much more care to avoid. Some precautions have been taken to avoid such errors. First, the specification is organized for *easy traceability*: clauses in the specification are cross-linked with particular page and line numbers of [82]. Second, the “formal semantic calculator” provided by the approach using familiar programming and debugging environments (e.g., Visual Studio) may help engage expert MPI users (who may not be formal methods experts) into experimenting with the semantic definitions.
```c
#include "mpi.h"

int main(int argc, char** argv)
{
  int rank, size, data1, data2, data3, flag;
  MPI_Request req1, req2, req3;
  MPI_Status stat;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if(rank == 0){
    data1 = 0;
    data2 = 0;
    MPI_Irecv(&data1, 1, MPI_INT, 1,
               0, MPI_COMM_WORLD, &req1);
    MPI_Irecv(&data2, 1, MPI_INT, 1,
               1, MPI_COMM_WORLD, &req2);
    MPI_Irecv(&data3, 1, MPI_INT, 1,
               2, MPI_COMM_WORLD, &req3);
  } else {
    data1 = 7;
    data2 = 6;
    MPI_Issend(&data1, 1, MPI_INT, 0,
                1, MPI_COMM_WORLD, &req1);
  }

  if(rank == 1){
    MPI_Wait(&req1, &stat);
    MPI_Irsend(&data2, 1, MPI_INT, 0,
                0, MPI_COMM_WORLD, &req2);
    MPI_Irsend(&data3, 1, MPI_INT, 0,
                2, MPI_COMM_WORLD, &req3);
  } else {
    MPI_Wait(&req2, &stat);
  }
  if(rank == 0){
    MPI_Wait(&req1, &stat);
  } else {
    MPI_Wait(&req2, &stat);
  }
  MPI_Finalize();
  return 0;
}
```

**Figure 1.2.** The C-based Litmus Test used to answer Question 4 on Page 4.
1.3 Organization of the Dissertation

This dissertation is organized as follows. Chapter 2 presents an overview of the TLA+ model of MPI. The full text of the model and a representative program model are included as Appendices A and B respectively. Chapter 3 selects a subset of MPI operations, simplifies the semantics of MPI to form a closed system for this subset (i.e., adding some functionality to some of the operations to make them work without the remainder of the model discussed in Chapter 2), proves a number of independence theorems for these operations, and presents a dynamic partial-order reduction algorithm that leverages these semantics. Chapter 4 discusses the model extraction framework, the abstraction based simplifications made to program models, and the simulation of error trails in the Visual Studio environment. Chapter 5 presents the design and implementation details of the MPIC model checker. MPIC implements the semantics and partial order reduction described in Chapter 3. Chapter 6 discusses future possibilities in this area and concludes.
CHAPTER 2

A SEMANTIC MODEL OF MPI

The natural language version of the Message Passing Interface (MPI) standard is a specification of the communication semantics of a software library. It describes the various operations that are included in a library that implements MPI and specifies the programming interface for FORTRAN, C, and C++. Some of the details, such as the values returned by function calls, under a variety of circumstances, are sketchy. However, the natural language document does give a comparatively good picture of how operations interact and what the nature of computation will be as a result of applying these operations.

Given as good a specification as the MPI standard is, it is desirable to describe aspects of the specification using mathematical precision, particularly where errors both in implementation and application of the standard may be manifested. Without a mathematical model of the MPI standard, one might conceive of a “golden implementation” that would serve as a reference implementation, from which others could determine correctness of proprietary implementations. The MPICH [32] MPI implementation may serve in this capacity as it is a free MPI implementation intended as a starting point for proprietary implementations. There are numerous reasons that a mathematical model, rather than a reference implementation, is desirable. In particular, a model allows one to abstract away implementation concerns while retaining the specified functional behaviors of all libraries that implement the MPI standard. In addition it allows us to state and prove properties of programs that use libraries implementing the MPI standard, without confining these proofs to a given MPI implementation — thereby aspiring to the portability tenets of the MPI standard.
This chapter is organized as follows. The Temporal Logic of Actions (TLA+) is introduced in Section 2.1. Section 2.2 presents an introduction of MPI. Section 2.3 gives an overview of the architecture of the MPI communication semantics model. Section 2.4 describes the MPI operations that are included in the model. The invariants and assertions placed in the model are discussed in Section 2.5. Exclusions from the model are detailed in Section 2.6. The modeling of message communication (Section 2.7), point-to-point message initiation primitives (Section 2.8), point-to-point message completion primitives (Section 2.9), and composite operations such as MPI_Send (Section 2.10) are discussed. The collective operations modeled are described in Section 2.11. The initial state of the model is described in Section 2.12. The chapter concludes with issues raised in the course of modeling in Section 2.13, related work in Section 2.14, and a summary of the chapter in Section 2.15.

\section{Introduction to TLA+}

The Temporal Logic of Actions (TLA+) is a formal mathematical notation proposed by Leslie Lamport [46]. TLA combines the expressiveness of first order logic with some temporal logic operators. Foremost among the temporal logic operators is the next state operator, denoted using \( ' \) or prime. For example:

\[ x' = [\mathbb{N} \rightarrow \mathbb{N}] \]

indicates that in the next time step (i.e., next state), the variable \( x \) is assigned to the set of all functions that map natural numbers onto natural numbers. Although TLA+ includes some temporal logic operators (such as the next state operator), having a familiarity with basic set theory is usually sufficient to enable understanding of specifications written in TLA+.

As TLA+ is used heavily throughout this chapter, it may be helpful to consider a more involved example, such as the dining philosophers [19] specification shown in Figure 2.1. For aesthetics the specification is divided into four sections. The first section includes a declaration of modules that are used by the philosophers module.
MODULE philosophers
EXTENDS Naturals, TLC
CONSTANT N  The number of forks and philosophers.

VARIABLE Forks, Phil_state
vars ≝ (Forks, Phil_state)
Type_invariant ≝
  \(\land Forks \in [0 . . (N - 1) \rightarrow 0 . . N]\)
  \(\land Phil_state \in [0 . . (N - 1) \rightarrow \{\text{"Eating"}, \text{"Thinking"}, \text{"Got One Fork"}\}]\)

Is fork \(f\) available?
Fork available\((f)\) ≝ Forks\([f]\) = N

Philosopher \(p\) acquires fork \(f\).
Fork acquire\((f, p)\) ≝ Forks′ = [Forks EXCEPT ![f] = p]

Philosopher \(p\) releases fork \(f\).
Release fork\((f)\) ≝ Forks′ = [Forks EXCEPT ![f] = N]

Philosopher \(p\) releases fork \(p\) and \((p + 1)\%N\).
Release forks\((p)\) ≝ Forks′ = [i \in 0 . . (N - 1) \mapsto
  IF i = p \lor i = (p + 1)\%N
  THEN N
  ELSE Forks[i]]

Phil Life\((\text{person})\) ≝ The (broken) life of a philosopher.
  \(\lor \land Phil\_state[\text{person}] = \text{"Thinking"}\)
  \(\land Fork\_available((\text{person} + 1)\%N)\)
  \(\land Fork\_acquire((\text{person} + 1)\%N, \text{person})\)
  \(\land Phil\_state′ = [Phil\_state \text{ EXCEPT ![}\text{person}] = \text{"Got One Fork"}]\)
  \(\lor \land Phil\_state[\text{person}] = \text{"Got One Fork"}\)
  \(\land IF Fork\_available(\text{person})\)
  \(\land \land Fork\_acquire(\text{person}, \text{person})\)
  \(\land Phil\_state′ = [Phil\_state \text{ EXCEPT ![}\text{person}] = \text{"Eating"}]\)
  \(\lor ELSE\land Release\_fork((\text{person} + 1)\%N)\)
  \(\land Phil\_state′ = [Phil\_state \text{ EXCEPT ![}\text{person}] = \text{"Thinking"}]\)
  \(\lor \land Phil\_state[\text{person}] = \text{"Eating"}\)
  \(\land Release\_forks(\text{person})\)
  \(\land Phil\_state′ = [Phil\_state \text{ EXCEPT ![}\text{person}] = \text{"Thinking"}]\)

Init ≝ \(\land Forks = [i \in 0 . . (N - 1) \mapsto N]\)
  \(\land Phil\_state = [i \in 0 . . (N - 1) \mapsto \text{"Thinking"}])\)

Next ≝ \(\exists p \in 0 . . (N - 1) : Phil\_Life(p)\)

Spec ≝ Init \(\land □[\text{Next}_\text{vars}]\)

Figure 2.1. A TLA+ specification of the classic Dining Philosophers problem.
In this case, Naturals and TLC (Naturals provides support for reasoning about expressions in $\mathbb{N}$ and TLC allows the specification to be model checked) are used. This is followed by a constant parameter $N$ and a comment about $N$ (line 3). Constants are given concrete values when the specification is model checked in a configuration file. Variables used in this specification and an invariant with respect to these variables comes next, in particular, that $\textit{Forks}$ is always a function from $\{n : 0 \leq n < N\}$ to $\{n : 0 \leq n \leq N\}$, and $\textit{Phil\_state}$ is similarly always a function as shown on line 8.

In addition to sets and functions, TLA+ allows users to specify TLA+ operators. An operator is similar to a predicate in first order logic. Operators may contain guards. When the guard evaluates to true, any applications of the next state operator are invoked, modifying the value of the specified state elements in the next state.

The second section of the philosophers module includes four operators. The first of these operators does not change any variable values and can therefore be used as a transition guard. The remaining three operators modify the value of the $\textit{Forks}$ state element by applying the next state operator. As $\textit{Forks}$ is a function, the TLA+ notation provides a shorthand means for specifying that only one or some of the elements of the range of that function have changed. Line 13 changes the value that $\textit{Forks}$ maps to under $f$ to $p$ in the next state. Lines 17 – 20 change both the values mapped by $p$ and $(p + 1)\%N$.

Section 3 of the specification (lines 22 – 35) contain a state machine, having three states and four transitions, modeling the broken life of a philosopher. This state machine is encoded as a logical formula where white space is significant; disjuncts and conjuncts that line up vertically are treated at the same level of precedence — eliminating the (sometimes confusing) parenthesis matching. When the philosopher is in the “Got One Fork” state, she checks to see whether the fork at index $\textit{person}$ is available. If so, she acquires the fork and proceeds to the “Eating” state. Otherwise she releases the fork at index $(\textit{person} + 1)\%N$ and resumes “Thinking”.

The final section of the example specifies the initial state of the system ($\textit{Init}$), the next state relation ($\textit{Next}$), and what the global state space of the system is
comprised of \((Spec)\). The \(Spec\) is defined using the temporal operator \(hensforth\), meaning in all future time steps and would read: “\(Spec\) is defined as Init and hensforth the Next state relation over vars.”

The primary reasons for using TLA+ for the model of MPI was the combination of formality and executability offered by the existing reasoning tools. Tools for reasoning about TLA+ specifications include the TLC explicit state model checker, and an embedding of TLA+ in the input language of the Isabelle theorem prover. Applying TLC to the specification of Figure 2.1 discovers the deadlock situation very quickly for a small number of philosophers.

In previous efforts to model the communication semantics of MPI, even with \(+CAL\) [67] which is automatically compiled into TLA+, the mathematical precision to which this work aspired was not sufficient. While model checkers exist for other languages such as Promela, the formal semantics of Promela are difficult to pin down. Moreover it became difficult for experts in MPI who were not experts in formal verification, or expert in the syntax of various modeling languages, such as \(+CAL\) and Promela to fully understand the models. This difficulty makes the more imperative modeling languages, such as Promela, \(+CAL\), and Zing, less attractive as a basis for formalizing MPI.

### 2.2 Introduction to MPI

This section presents an introduction to MPI. Several good books (e.g., [42, 64, 80]) far exceed this presentation in breadth of content and level of detail. An attempt to demystify the naming convention of MPI point-to-point operations is presented, followed by a very simple MPI-based C program as an example.

#### 2.2.1 A Description of the MPI Primitive Naming Convention

MPI has a naming convention that allows users to understand the intended semantics of a communication primitive based on the sequence of letters that are part of the operation name. The variants are Blocking mode (having no character prefix), Immediate mode (represented by an \(I\) after the \(MPI\_\) and before the send or receive designation), and Persistent mode (where the request creation ends with \(_\text{init}\)).
Blocking mode operations cause the call to block the sender until the corresponding requirement has been fulfilled with respect to the type of the operation. Immediate mode operations do not block the caller, rather a request handle is created and returned to the caller who must eventually use `MPI_Wait` or `MPI_Test` or some variant to insure the operation has completed before the memory area can be accessed. Persistent mode operations create a request, just as the immediate mode operations; however the request is not deallocated after the request completes and can therefore be reused.

Within this set of variants, send operations can be Ready, Synchronous, Buffered, or Standard. The send operation variant is also encoded using a character in the operation name: `r` or `R` for Ready send, `s` or `S` for Synchronous send, `b` or `B` for buffered send, and no prefix character for Standard send. Ready mode sends require the corresponding receive to have been posted prior to the call of the Ready mode send. The synchronous mode send instructs the MPI communication subsystem to block the completion of the corresponding operation until the memory can be reused safely without using buffering. Buffered mode sends instruct the MPI communication subsystem to buffer the send using a specific memory area provided by a call to `MPI_Buffer_attach`. Standard mode sends may be buffered by the MPI communication subsystem if buffering is available, however buffering is not guaranteed so operations using this send mode may cause synchronization. As an example, the variants of Buffered send are `MPI_Bsend` (blocking the calling process until the message is buffered or transmitted), `MPI_Ibsend` (allowing the message to complete when the message is buffered or transmitted), and `MPI_Bsend_init` (creating an inactive request for a buffered send).

Variants of `MPI_Wait` and `MPI_Test` allow for blocking or non-blocking message completion on arrays of requests, requiring all, some, or one of the requests in the array to complete. For a complete description, the interested reader should see [82].

### 2.2.2 A Simple Example of an MPI Program

To give the reader some flavor for the style and structure of an MPI program, in this section the simplest of “Hello World” examples is presented, shown in
Figure 2.2. This program passes a token between processes in order based on rank. The process prints their rank and “Hello World!” just before it sends the token.

A process must first call MPI_Init to communicate using a library that implements MPI. This program is written in the SPMD style; every process in the computation executes the same program image. They are able to differentiate based upon the rank of the process which is assigned by a call to MPI_Comm_rank. Ranks are assigned consecutive values from the natural numbers, starting with rank 0. The call to MPI_Comm_size writes the number of processes that are participating into the size variable on Line 5.

Messages can be sent and received using ranks to compute the address of the recipient or sender. This program causes one process, the process with rank 0, to send the token and then receive the token. All other processes receive the token

```c
void main(int argc, int** argv){
    int rank, size, token;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(rank == 0){
        printf('%d: Hello World!
',rank);
        MPI_Send(&token,1,MPI_INT,((rank+size)+1)%size,
                 0,MPI_COMM_WORLD);
        MPI_Recv(&token,1,MPI_INT,((rank+size)-1)%size,
                 0,MPI_COMM_WORLD);
    } else {
        MPI_Recv(&token,1,MPI_INT,((rank+size)-1)%size,
                 0,MPI_COMM_WORLD);
        printf('%d: Hello World!
',rank);
        MPI_Send(&token,1,MPI_INT,((rank+size)+1)%size,
                 0,MPI_COMM_WORLD);
    }
    MPI_Finalize();
}
```

Figure 2.2. A simple “Hello World” example C program using MPI.
and then send. This asymmetric behavior avoids potential deadlock. After the
token has passed a particular process, that process is ready to terminate. The
call to MPI_Finalize indicates to the MPI library that no more communication
operations will be execution by that particular process.

2.3 Overview of MPI Model Architecture

The TLA+ model of MPI is intended to capture details that are both im-
plied and explicitly referenced in the natural language standard with mathematical
precision, while abstracting away the implementation specific issues that are not
specified. The model broadly implements the architecture shown in Figure 2.3.
The specification API is preserved such that application of an MPI operation has
a similar external interface as an MPI procedure call in C. The main pieces of the
model are point to point operations, collective operations, and constants.

Point to point and collective operations are coupled using a communicator. The
communicator is used to isolate the interactions of processes. Every MPI commu-
nication operation requires reference to a communicator. Using the first order logic

Figure 2.3. The architecture of the TLA+ MPI model.
facilities of TLA+, a communicator is a function from handles \((\text{handles} \in \mathbb{N})\) to a tuple containing a group and a context. The model of the communicator is restricted as MPI additionally has topologies and attributes, which is considered future work. The context houses all information about messages that are currently available for communication within the communicator. Messages within a communicator are referred to by processes using process ranks. A group defines the set of processes allowed to access messages within a communicator and their respective ranks.

2.4 Modeling Approach

The MPI 1.1 standard [82] contains 128 operations that provide a rich collection of communication options. A full 35 of these operations are dedicated to pair-wise exchanges of messages between processes. The model contains those operations representable using exactly one TLA+ atomic transition (primed variables equated to unprimed variables). The operations included are shown in Table 2.1. The remaining seven operations can be modeled as sequential compositions of those shown in Table 2.1. Thus \texttt{MPI\_Send} becomes \texttt{MPI\_Isend} and \texttt{MPI\_Wait} issued in that order. Similarly, \texttt{MPI\_Sendrecv} becomes \texttt{MPI\_Isend}, \texttt{MPI\_Irecv}, and two \texttt{MPI\_Wait} operations issued sequentially; and so on.

The reason for using the relatively standard technique of composition for these operations is to eliminate the excessive additional overhead that would be necessary.

<table>
<thead>
<tr>
<th>\texttt{MPI_Get_count}</th>
<th>\texttt{MPI_Request_free}</th>
<th>\texttt{MPI_Test_canceled}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{MPI_Buffer_attach}</td>
<td>\texttt{MPI_Waitany}</td>
<td>\texttt{MPI_Send_init}</td>
</tr>
<tr>
<td>\texttt{MPI_Buffer_detach}</td>
<td>\texttt{MPI_Testany}</td>
<td>\texttt{MPI_Bsend_init}</td>
</tr>
<tr>
<td>\texttt{MPI_Isend}</td>
<td>\texttt{MPI_Waitall}</td>
<td>\texttt{MPI_Ssend_init}</td>
</tr>
<tr>
<td>\texttt{MPI_Ibsend}</td>
<td>\texttt{MPI_Testall}</td>
<td>\texttt{MPI_Rsend_init}</td>
</tr>
<tr>
<td>\texttt{MPI_Issend}</td>
<td>\texttt{MPI_Waitsome}</td>
<td>\texttt{MPI_Recv_init}</td>
</tr>
<tr>
<td>\texttt{MPI_Irsend}</td>
<td>\texttt{MPI_Testsome}</td>
<td>\texttt{MPI_Start}</td>
</tr>
<tr>
<td>\texttt{MPI_Irecv}</td>
<td>\texttt{MPI_Iprobe}</td>
<td>\texttt{MPI_Startall}</td>
</tr>
<tr>
<td>\texttt{MPI_Wait}</td>
<td>\texttt{MPI_Probe}</td>
<td></td>
</tr>
<tr>
<td>\texttt{MPI_Test}</td>
<td>\texttt{MPI_Cancel}</td>
<td></td>
</tr>
</tbody>
</table>
to facilitate modeling them directly. For example, consider the additional information needed to model `MPI_Ssend` directly. For each process, a map from the program counter (pc) to the next operation to be performed when `MPI_Ssend` is enabled would be required. In this manner, it can be determine when a corresponding `MPI_Recv` could be executed by the receiving process, and then cause both processes to jointly execute their state transition steps. However, since there is no restriction on what type of receive could be matched with `MPI_Ssend` (it could be `MPI_Recv`, `MPI_Irecv`, `MPI_Sendrecv`, etc.), nor are there restrictions on the blocking nature of the receives (some block the receiving process while others do not), supporting each of the variants becomes quite laborious, in addition to resulting in unreadable model descriptions.

Additional supporting operations included in the model are shown in Table 2.2. Each of the operations has the same parameters in the same order as the MPI standard, with two additions. First, there is no way for TLA+ operations to query the system to discover which process is executing, short of having a globally visible state element. Therefore, the PID of the process executing an MPI call is passed as a parameter, which appears after the parameters specified in the standard. There is no graceful way to provide return values of MPI function calls. The return address is, therefore, also provided as a parameter (although handling return values remains future work).

**Table 2.2.** The additional MPI operations modeled to enable model checking based reasoning on MPI-based parallel programs.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Parameter 1</th>
<th>Parameter 2</th>
<th>Parameter 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>MPI_Group_size</td>
<td>MPI_Group_rank</td>
<td></td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>MPI_Comm_rank</td>
<td>MPI_Comm_compare</td>
<td></td>
</tr>
<tr>
<td>MPI_Init</td>
<td>MPI_Finalize</td>
<td>MPI_Initialized</td>
<td></td>
</tr>
<tr>
<td>MPI_Abort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.4.1 Data Structures in the Model

The data structures that are included in the model represent the state of the MPI system. These structures are communicators, groups, requests, messages, and statuses. All of the MPI specified constants are modeled as TLA+ constants. Each must be given a value in the TLC configuration file to facilitate model checking. The remainder of this subsection discusses each of the data structures, as modeled, in detail.

2.4.1.1 Communicators

Communicators provide a boundary between the interactions of individual communication operations. The standard makes clear that handles are to be opaque (not user program accessible) objects that are held by a given process in reference to various communicators. This implies that the same handle value on a different process may in fact reference a different communicator. A communicator is modeled as a function from handles to records (a record is just a function) where the fields are group and collective contexts (described in Sections 2.4.1.2 and 2.4.1.3 respectively). Each process may map handles onto these communicator records differently. There is a communicator function for each process (i.e., no sharing between processes). The communicator structures are grouped together thus a single function from process identifiers to communicator functions.

As a simplification, the communicator handles are currently unique — meaning that if handle for MPI_COMM_WORLD has the value of 7 on some process, then that handle has the value 7 on all processes. When the model is extended to include additional operations on groups and communicators it will be necessary to refine this detail of communicators.

2.4.1.2 Groups

A group is also represented by a function from communicator handle to records having a set of process identifiers as group members, the number of processes in the group, and the ranking and inverse ranking functions for the group. It is this ranking function that is used to determine the rank of a calling process when
MPI_Comm_rank is applied. A process must be a member of the group in order to participate in communications within the communicator. As with communicators, each process maintains its own group function.

### 2.4.1.3 Collective Context

The collective context is used to retain information regarding collective operations. This model abstracts most data (as described in Section 2.6). The collective context, modeled as a function from communicator handles to records, retains the set of processes participating in the collective operation, the root process, the type of collective operation, and the collective state. The modeled collective operations are described in Section 2.11. The array (function) of collective contexts is indexed by a unique identifier contained in the communicator that references the given collective context. There need be only one collective context for each communicator as the standard does not allow for out of order execution of collective operations on different processes.

### 2.4.1.4 Requests

Requests are the data that is used in addition to the message envelope described in Section 2.4.1.5 to form communication between processes. The request contains a set of fields indicating the current state of a request, whether the request has completed, an error code, and the actual message. If the message has been paired with another request for transmission (see Section 2.7.1 for a description of pairing), the match field is changed from the empty pair to a pair representing the process identifier and index on that process of the request that is paired.

The bit flags are used to implement a protocol for determining when the message can and has completed. The message has completed when either:

- The message has been canceled. This is possible using `MPI_Cancel` for immediate mode and persistent send or receive operations.

- The message has been buffered. There are two cases when the message could be buffered: (i) the message was initiated using `MPI_Send` or a variant; the
system provides buffering, and (ii) the message was initiated using `MPI_Bsend` or a variant; the user provides buffering.

- The message has been successfully transmitted.

The transmit, buffer, and cancel operations are further discussed in Sections 2.7.2, 2.7.3, and 2.7.4 respectively.

Individual requests are maintained in a sequence on each processes (again a function, \( \text{requests} : \text{pid} \to \mathbb{N} \to \text{request} \)) where the elements of the request sequence domains are greater than zero. To add a request to the sequence, the least number in \( \mathbb{N} \notin \text{Dom}(\text{requests}) \) is added to \( \text{Dom}(\text{requests}) \) in the next state, with the new request mapped appropriately. In the model, no request is ever removed from the requests sequence.

Requests can be persistent when created using one of the `MPI_Send_init`, `MPI_Ssend_init`, `MPI_Bsend_init`, `MPI_Rsend_init`, or `MPI_Recv_init` operations. This is modeled by setting the appropriate flag in the request record. Persistence means the request is not deallocated when the completing operation returns, allowing the process to retain the inactive request handle and again use the request for communication by optionally changing the contents of the message buffer and again calling `MPI_Start` on the handle.

### 2.4.1.5 Messages

Messages represent the envelope and the data address passed in by a message initiation operation (both send and receive). Each message record contains the source rank, destination rank, user defined message tag, user specified MPI data type, number of elements (count), and communicator handle. The memory address of the data being transmitted, and whether the message was created by a send or receive operation, is also recorded.

### 2.4.1.6 Statuses

Most of the point to point operations return some information to the calling process in the status structure. Unless using `MPI_Probe`, `MPI_Iprobe`, or `MPI_Cancel`
this information is typically ignored by the program. Information produced by these operations is contained in a status structure. The MPI standard allows data members of this structure to be accessed directly.

The model has a fairly standard notion of process memory (described in Section 4.1.2), as a function Memory : \mathbb{N} \to \mathbb{N} where user variables are allocated and manipulated. The status structure of MPI requires five integer fields, three of which are user accessible. Status structures are modeled by allocating five consecutive memory locations in the process memory. Status structures are accessed using a base address and a given offset for the particular field.

2.5 Invariants and Assertions Included in the TLA+ MPI Model

The TLA+ MPI model contains properties in the form of invariants and assertions. The invariants essentially comprise a simple type system that requires data values to be kept within appropriate ranges and records to be of a specified structure. Thus invariants in the MPI model assert things such as “only request structures are added to the request sequence of a process.”

The TLA+ MPI model has several assertions that are placed with associated MPI operations. An assertion is an expression over the state variables that must evaluate to true whenever the guard for the accompanying transition also evaluates to true. The remainder of this section lists the assertions that were placed in the model in addition to the invariants.

The MPI standard requires the following constraints to be met by all MPI operations:

- No MPI operation, except MPI_Initialized, can be applied by a process until MPI_Init has been called by that process.

- No MPI operation can be applied by a process after MPI_Finalize has been called by that process.

- A process addressed by a message must be a member of the group for the communicator used in the creation of that message.
• All processes must eventually call MPI_Finalize.

The requirement that all processes must eventually call MPI_Finalize is not equivalent to process termination. It is possible that a process may continue to execute and perform additional computation after calling MPI_Finalize. However, it is assumed this is not a useful situation and require all processes to eventually terminate. This requirement is enforced by checking that all processes eventually reach a known final state.

There are also specific assertions made about individual operations. For convenience, all of the assertions included in the model are presented. These are assertions checked during state enumeration whenever the corresponding MPI primitive is applied.

• The count of a request that has been canceled is never read.

• Processes cannot attach more than one buffer using MPI_Buffer_attach in any state.

• A buffer must have been attached before a call to MPI_Buffer_detach.

• Buffer space must be sufficient when calling MPI_Bsend and variants.

• When using MPI_Rsend and variants, there must exist a corresponding receive request on the specified receiving process.

• A call to MPI_Request_free cannot be made on a request that is a receive.

• A call to MPI_Request_free should not be made unless the request has either (i) completed locally or (ii) completed locally on the receiving process.

• A call to MPI_Start and MPI_Startall cannot have a request that is currently active or null, each request representing a MPI_Rsend or variant must satisfy the ready send requirements when the request is started, and if the request houses a MPI_Bsend variant then there must be sufficient buffering available when the request is started.
• A call to MPI_Start and MPI_Startall must only reference persistent requests.

• When MPI_Finalize is called, the user provided buffers have been detached. Detaching the buffer causes the calling process to block until all messages that are currently buffered have transmitted.

• When MPI_Finalize is called, there are no locally active requests on the calling process. When combined with the requirement that all processes must eventually become finalized implies that every message posted is eventually canceled, transmitted, or explicitly deallocated.

2.6 Abstractions and Simplifications Imposed on the TLA+ MPI Model

It is important to point out that not modeled all of the semantics of the MPI 1.1 standard are present in this model. The following lists several of the abstractions and assumptions made with regards to MPI, the programs that use MPI, and the types of properties that can be proved using model checking with the model.

• Most data is abstracted away. Data, such as arrays of floating point values, objects, etc., could be modeled using TLA+. It is, however, not necessary in most cases to retain the actual data values of a distributed computation to verify a large class of interesting reactive properties. Therefore a placeholder for data in the formal model is present in such a way that data can be included when necessary. The model does allow for the preservation of one integer data value in each message. This value can be passed between processes using the MPI operations.

• Operations other than those listed in Table 2.2 on communicators and topologies are not used. Operations on communicators and topologies are modeled to a limited extent to enable point to point communications on intra-communicators. The operations shown in Table 2.2 are included in the model,
in addition to the point-to-point operations of Chapter 3 of MPI 1.1 shown in Table 2.1.

- Implementation details. To the greatest extent possible the model avoids asserting implementation-specific details. One obvious ramification of this omission is that modeling return codes of MPI operations is completely eliminated (cf. [82, Page 11]).

- Handling implementation-dependent buffer availability. As far as the standard mode sends (e.g., `MPI_Send`, `MPI_Isend`, `MPI_Send_init`) go, the model requires the system to either eventually buffer these requests or to not buffer them at all. The standard allows for an implementation to switch between these policies in a time-varying manner; attaining such generality would complicate the semantics drastically.

- User provided buffering. Buffering is abstracted such that a counting semaphore tracks the number of messages that can be buffered using the explicit space provided by the user — rather than modeling the number of bytes being sent per message.

- The progress requirement of `MPI_Test`. The text description of `MPI_Test` requires that if `MPI_Test` is called repeatedly and there is a request that could match, that the `MPI_Test` will eventually succeed. The variants of `MPI_Test` included do not fulfill this requirement.

2.7 Model Granularity Issues

A formal model for a communications library must model at the right level of granularity in order to not mask corner cases. To achieve this objective, three additional rules are introduced that are allowed to interleave with the actions of an individual process.

As mentioned in Section 2.4.1.4 a message completes locally (on the calling process) when that message has been canceled, buffered, or transmitted. Granularity is determined in the model by the effects of the individual MPI operations that must
become observable due to the ways in which MPI operations interact. The model in Chapter 3 has a much coarser granularity due to the reduction in operations. In particular, there are no buffering send or multiple completion operations — thereby causing a great simplification in the model and a reduction in the number of states that are generated when model checking with respect to that model. In the most general setting for MPI, meaning all of the operations must be able to interact, rules which facilitate message pairing, message buffering, and message transmission are sufficient to observe the possibility space for the different interactions.

2.7.1 Matching and Pairing Messages

Messages are paired together using quantification over the request structures of two processes. Requests should Match, be globally active, started, not canceled, not transmitted, one be a send, the other a receive, and both have not been paired previously.

The predicate Match is defined on two message records. Two messages are said to match if all of the following are true: (i) the destinations are equal, (ii) the sources are equal or the source of the receive message is MPI_ANY_SOURCE, (iii) the types are equal, (iv) the tags are equal or one of the tags is MPI_ANY_TAG, and (v) the communicators are equal.

In addition, the paired requests, both sending and receiving, must be the least requests in the sequences in the order that messages were added to the request sequences. This is reflected in the value of the corresponding request index — i.e., earlier requests in program order have lower request indices on the host process. The reader is referred to Appendix A, lines 454 – 533 for the implementation of these rules.

2.7.2 Transmitting Messages

Figure 2.4 shows the TLA+ model of the Transmit rule. This rule is allowed to execute asynchronously with the processes being verified and the other rules of this section. This rule is enabled when there exists process \( i \), and request \( j \) on process \( i \) such that the request is started, is globally active, has not been canceled, has not
module Transmit

∀ i ∈ 0..(N − 1) :

∀ j ∈ 1..Len(requests[i]) :

let m = requests[i][j] in

∧ m.started The message has been started.
∧ ¬m.canceled The message has not been canceled.
∧ ¬m.transmitted The message has not been transmitted.
∧ ∨ m.match ≠ ⟨⟩ Either the message has been matched or

∨ m.message.src = MPI_PROC_NULL 61.1 the sender is null or

∨ m.message.dest = MPI_PROC_NULL the receiver is null.

∧ requests' = [requests EXCEPT ![i] = Mark the request transmitted.

[@ EXCEPT ![j] =

[@ EXCEPT !.transmitted = TRUE]]]

∧ IF m.match ≠ ⟨⟩ ∧ ¬requests[m.match[1]][m.match[2]].transmitted THEN Need to move the data.

IF m.message.state = "recv"

THEN Write the data from the sender into i’s memory.

Memory' =

[Memory EXCEPT ![i] =

[@ EXCEPT ![m.message.addr] =

Memory

[m.match[1]]

[requests[m.match[1]][m.match[2]].message.addr]]]

ELSE Write the data from i into the receiver’s memory.

Memory' =

[Memory EXCEPT ![m.match[1]] =

[@ EXCEPT ![requests[m.match[1]][m.match[2]].message.addr] =

Memory[i][m.message.addr]]]

ELSE

UNCHANGED ⟨Memory⟩

∧ IF m.cctype = "bsend"

THEN Need to release the buffering.

message_buffer' = [message_buffer EXCEPT ![i] = @ − 1]

ELSE

UNCHANGED ⟨message_buffer⟩

∧ UNCHANGED ⟨group, communicator, bsize, initialized, collective⟩

Figure 2.4. The Transmit rule for moving data between processes in the TLA+ MPI model.
been transmitted, and has already been paired with another request on some other process. It is necessary to pair and transmit messages separately because there is no requirement for ordering of message completion in the MPI standard [82]. Consider the case where two messages are sent from process 1 to process 2 where the first message is very large and the second message is very small. The MPI standard requires that the first message sent be matched with the first receive posted in program order on both processes. However this makes no statement about when the messages will complete. In this example, it should be possible for the smaller message to complete first. The use of a separate transmit rule allows us to facilitate the modeling of MPI_Cancel which is used to cancel pending MPI messages. Further discussions on the issues raised while modeling are provided in Section 2.13.

Continuing with Figure 2.4, the final three conjuncts in the Transmit rule define the values of Memory, requests, and the message_buffer in the next state. In MPI, the event marking the completion of the transmission on the sender side may become visible before the event on the receiver side, or vice versa. Therefore, only one request is updated to show that the transmitting step has completed.

2.7.3 Buffering Messages

The model interleaves the Buffer rule asynchronously, allowing messages that can be buffered to nondeterministically be completed locally at any state before the message is transmitted. Existential quantification is used to specify some request on some process where buffering can take place and mark that the buffering is accomplished. The value at the address specified is copied into a temporary location so the memory can be safely reused by the process. This rule is located on lines 584 – 611 of Appendix A.

2.7.4 Canceling a Message

The MPI standard requires that the cancel operation MPI_Cancel on a request is always enabled and may, or may not, cause the data to not be transmitted; i.e., a canceled message may be successfully transmitted anyway. Nondeterministic scheduling discussed in Section 2.7.5 is used to model this feature. The MPI_Cancel
operator simply changes the value of the canceled flag in the request.

### 2.7.5 Full Transition Relation

The full transition relation defines the actions that can take place at any step of the model. Shown in Figure 2.5 the Pair, Transmit, and Buffer rules are allowed to execute whenever enabled. The processes are selected nondeterministically and allowed to execute when enabled, or all processes must have come to the end of their transition relation and are in a known final state.

### 2.8 Initiating Point to Point Messages

The bulk of the model comprises operations to initiate and complete messages. This section focuses on initiating messages by creating requests and adding those requests to the request sequence of the calling process. Section 2.10 describes how the blocking mode operations are modeled. The remainder of this section discusses creating and completing requests via Immediate and Persistent mode operations.

```plaintext
1 module Next
2 Next ≜
3 ∨ ∧ Pair Attempt to match two messages.
4 ∧ UNCHANGED program_vars
5 ∨ ∧ Transmit Attempt to transmit a message.
6 ∧ UNCHANGED ⟨Map, pc⟩
7 ∨ ∧ Buffer Attempt to buffer a message.
8 ∧ UNCHANGED program_vars
9 ∨ ∃ p ∈ 0 .. (N − 1) : Proc(p) Prevent spurious deadlocks.
10 ∨ ∀ p ∈ 0 .. (N − 1) : pc[p] = Final_State Insure all processes call MPI_Finalize.
11 ∧ ∀ p ∈ 0 .. (N − 1) : initialized[p] = “finalized”
12 ∧ UNCHANGED program_vars
13 ∧ UNCHANGED mpi_vars

Figure 2.5. The top level next-state relation for the TLA+ MPI model.
```
2.8.1 Immediate Mode Operations

Immediate mode operations include MPI_Isend, MPI_Issend, MPI_Ibsend, MPI_Irsend, and MPI_Irecv. Each of these operations creates a message envelope from the required API parameters. The message envelope is inserted into a request and the request is appended to the request sequence for the calling process. The flags in the request are set depending upon the operation variant used. The new request is not marked to persist and the request is globally and locally active when added to the request sequence. The index of the new request is also written into the handle’s memory location.

2.8.2 Persistent Mode Operations

Persistent operations are almost equivalent to those described in Section 2.8.1, with two important exceptions. When the request is created, it is marked as persistent, and the request is not marked as locally or globally active. Thus an additional call to MPI_Start or MPI_Startall is required before these requests can be buffered, paired, transmitted, or used to satisfy the ready mode send requirements.

2.9 Completing Point to Point Messages

When a request has been canceled, transmitted, or buffered, that message can be completed locally. The operations that are used to locally complete requests are MPI_Wait, MPI_Waitall, MPI_Waitany, MPI_Waitsome, MPI_Test, MPI_Testall, MPI_Testany, MPI_Testsome. Although all of these are included in the model, only MPI_Wait is shown in this section. Variations to accomplish the other operations are then discussed.

2.9.1 Waiting for Completion

Figures 2.6 and 2.7 contain the TLA+ model definition of MPI_Wait, commonly used to complete immediate and persistent mode communications. As with all MPI operations (except for MPI_Init), MPI_Init must have been called prior to the application of this operation. This is checked as an assertion on line 4 of the
module Wait

MPI_Wait(request, status, return, proc) =

let r = requests[proc][Memory[proc][request]] in

"Error: MPI_Wait called with proc not in initialized state.")

∆ =

let r = requests[proc][Memory[proc][request]] in

4 ∧ Assert(initialized[proc] = "initialized", 200.10-200.12
"Error: MPI_Wait called with proc not in initialized state.")

4 ∧ Assert(r.request ≠ MPI_REQUEST_NULL)

41.32 - Blocks until complete

∧ ∨ r.transmitted The message was transmitted or

∧ ∨ r.canceled canceled by the user program or

∧ ∨ r.buffered buffered by the system

∧ Memory' =

\[\text{Memory except } !\{\text{proc}\} = 41.36\]

[\@ except ![Status_Canceled(status)] =

∧ r.canceled

∧ ¬r.transmitted, 54.46

![Status_Count(status)] = r.message.numelements,

![Status_Source(status)] = r.message.src,

![Status_Tag(status)] = r.message.msgtag,

![Status_Err(status)] = r.error,

![request] = 41.32-41.35, 58.34-58.35

IF r.persist

THEN @

ELSE MPI_REQUEST_NULL]]

∨ ∧ ∨ r.message.src ≠ MPI_PROC_NULL

∨ r.message.dest ≠ MPI_PROC_NULL

∧ Memory' = [Memory except !\{\text{proc}\} = 41.36]

[\@ except ![Status_Canceled(status)] = r.canceled,

![Status_Count(status)] = 0,

![Status_Source(status)] = MPI_PROC_NULL,

![Status_Tag(status)] = MPI_ANY_TAG,

![Status_Err(status)] = 0,

![request] = 41.32-41.35, 58.34-58.35

IF r.persist

THEN @

ELSE MPI_REQUEST_NULL]]

∧ requests' =

IF r.match ≠ ()

THEN

[requests except !\{\text{proc}\} = 58.34

Figure 2.6. The TLA+ model of MPI_Wait, part 1.
Figure 2.7. The TLA+ model of MPI\_Wait, part 2
operation. The comments are of two types: regular and cross references into the
natural language version of the standard. The cross references are numbered as “page.line” following the TLA+ comments shown in gray boxes, and allow assertions to be traced back to the natural language MPI standard document. A few aspects of the specification of MPI_Wait are now examined.

The main conjunct in the specification causes the group, communicator, bufsize, message_buffer, initialized, and finalized to remain unchanged in the next state. It then considers two cases: when the request is the special constant MPI_REQUEST_NULL, or when it is a non-null request handle. For the non-null case, the operation becomes enabled when (i) the request is locally active — meaning it has not been previously completed by some wait or test variant, and (ii) the request indicates that the message has been transmitted, canceled, or buffered. In this case, if the source and destination referenced in the request are non-null, the memory of the executing process is updated to indicate that the message has completed by filling the fields of the status object (lines 15 – 27). Otherwise, the status fields are set to reflect that the completion has occurred on a request referencing MPI_PROC_NULL. In either case the request handle is appropriately set, and the status fields in memory are also marked.

The request sequence for the executing process must also be updated (lines 40 – 70 of Figures 2.6 and 2.7). When a communication between processes i and j is initiated by i using a buffering send (such as MPI_Send) or when using MPI_Cancel, it is possible for the MPI_Wait to become enabled before the matching request is posted on process j. This is apparent when r.match = ⟨⟩ on line 41. In the true case, the previously paired request is marked globally inactive, in addition to the local request being marked locally inactive and globally inactive. In the false case, only the local request is marked locally inactive. Again, the status fields are marked as required by the standard.

2.9.2 Testing for Completion

The MPI_Test operation is different from MPI_Wait in the following ways: MPI_Test does not block and MPI_Test returns a flag value to indicate whether
the message was completed. The flag returns true exactly when a MPI_Wait would have been enabled on the same request.

### 2.9.3 Variants: All, Any, and Some

Arrays are introduced to accommodate the all, any, and some variants of MPI_Wait and MPI_Test. An array of request handles are passed in contiguous process memory locations to the operation and an array of statuses containing the returned information regarding which of the operations completed is populated — again in the process memory — and returned to the calling process.

### 2.10 Compositional Modeling of Standard Mode Operations

Multi-step MPI operators can be compiled into some sequence of single-step operators. Possible solutions for the seven contained in the MPI standard that are not present in the TLA+ model are presented.

The MPI operations can be modeled using a sequence of transitions with proc being the PID of the process, ‘‘in’’ being the starting pc of the call to MPI_Send, ‘‘intermediate-i’’ being the introduced intermediate pc value, and ‘‘out’’ being the return pc.

These transformations are believed to be semantically preserving for the following reasons. These transformations have been discussed with the main implementors of the MPICH MPI distribution, who concur with this belief. Second, the multiple transition sequence is executed consecutively by the process, making the externally observable behavior stutter equivalent. Third, additional execution interleaving orders produced by these sequences of transitions will result in only redundant, property invisible, execution traces since transitions in these sequences will touch only the state specified in the original operation and temporary variables introduced by the transformation.
2.10.1 Models of the MPI_Send, MPI_Bsend, MPI_Ssend, MPI_Rsend, and MPI_Recv Primitives

Applying the composition technique MPI_Send in a program having sequential execution can be specified by exchanging the MPI_Send for a MPI_Isend followed immediately by a MPI_Wait on the resulting request handle. This transformation is shown using a pseudo-TLA+ form shown in Figure 2.8. Each of the blocking-mode operations can be similarly transformed. The buffering requirements and previously posted receive requirements are unchanged and not observable externally.

In Figure 2.8 the fields buf, req, and status must be memory locations in the Memory array. The first disjunct checks the pc as a guard, and sets the pc in the next state to an introduced intermediate value while applying the MPI_Isend operator to the current state. The second disjunct checks the pc as a guard, requiring pc to be the introduced intermediate value, sets the pc to the next state, and applies MPI_Wait to the current state. Temporary variables must also be introduced to house the request and status values that would otherwise be unnecessary.

The same template transformation can be applied to each of the blocking mode send and operations and the blocking mode receive. Simply replace the Send with

\[
\begin{align*}
\forall \ \text{pc}[\text{proc}] = \text{‘in’} \\
\forall \ \text{pc} = [\text{pc EXCEPT !}[\text{proc}] = \text{‘out’}] \\
\forall \ \text{MPI_Send}(&\text{buf, count, datatype, dest, tag, com, return, proc})
\end{align*}
\]

is transformed to

\[
\begin{align*}
\forall \ \text{pc}[\text{proc}] = \text{‘in’} \\
\forall \ \text{pc} = [\text{pc EXCEPT !}[\text{proc}] = \text{‘intermediate’}] \\
\forall \ \text{MPI_Isend}(&\text{buf, count, datatype, dest, tag, com, req, return, proc}) \\
\forall \ \text{pc}[\text{proc}] = \text{‘intermediate’} \\
\forall \ \text{pc} = [\text{pc EXCEPT !}[\text{proc}] = \text{‘out’}] \\
\forall \ \text{MPI_Wait}(&\text{req, status, return, proc})
\end{align*}
\]

Figure 2.8. A template for transforming MPI_Send into an MPI_Isend followed by a MPI_Wait.
Recv and the Isend with Irecev in Figure 2.8 to effect the appropriate transformation.

2.10.2 A Model of the MPI_Sendrecv Primitive

To facilitate the MPI_Sendrecv operation the transformation shown in Figure 2.9 is proposed. This transformation replaces the MPI_Sendrecv with a corresponding MPI_Isend and MPI_Irecv followed by two MPI_Wait operations. The transformation preserves everything but the return value return and the status value status. However, the TLA+ MPI model does not account for implementation specific return values and the status values are not specified by the MPI standard so the status structure values are filled by the MPI_Wait on the recvreq.

```plaintext
\| \ /
\| pc[proc] = 'in'
\| pc' = [pc EXCEPT ![proc] = 'out']
\| MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status, return, proc)

is transformed to

\| \ /
\| pc[proc] = 'in'
\| pc' = [pc EXCEPT ![proc] = 'intermediate-1']
\| MPI_Isend(sendbuf, sendcount, sendtype, dest, sendtag, comm, sendreq, return, proc)
\| \ /
\| pc[proc] = 'intermediate-1'
\| pc' = [pc EXCEPT ![proc] = 'intermediate-2']
\| MPI_Irecv(recvbuf, recvcount, recvtype, source, recvtag, comm, recvreq, return, proc)
\| \ /
\| pc[proc] = 'intermediate-2'
\| pc' = [pc EXCEPT ![proc] = 'intermediate-3']
\| MPI_Wait(sendreq, status, return, proc)
\| \ /
\| pc[proc] = 'intermediate-3'
\| pc' = [pc EXCEPT ![proc] = 'out']
\| MPI_Wait(recvreq, status, return, proc)
```

Figure 2.9. A template for transforming MPI_Sendrecv into an MPI_Isend and an MPI_Irecv transition followed by a pair of MPI_Wait transitions.
2.10.3 A Model of the MPI_Sendrecv_replace Primitive

To facilitate the transformation of the MPI_Sendrecv_replace operation, the transformation shown in Figure 2.10 is proposed. This transformation is similar to that shown in Figure 2.9, with the addition of a temporary buffer recvbuf into which the message is read and then copied to the send buffer sendbuf after the corresponding MPI_Wait has returned.

\[ \text{MPI Sendrecv replace} \]

\[ \text{MPI Sendrecv} \]

\[ \text{MPI Isend} \]

\[ \text{MPI Irecv} \]

\[ \text{MPI Wait} \]

\[ \text{Assignment} \]

Figure 2.10. A template for transforming MPI_Sendrecv_replace into an MPI_Isend and an MPI_Irecv transition followed by a pair of MPI_Wait transitions and an assignment.
2.11 Collective Operations Included in the Model

Any collective operation can be synchronizing according to the MPI standard. With the restricted notion of data imposed on this model, only a model of the `MPI_Barrier` is included. To specify `MPI_Barrier` the presence of a collective context is required as described in Section 2.4.1.3. The actual barrier is specified using two transitions, `Barrier_init` and `Barrier_wait`.

The barrier is modeled as a synchronization protocol on the collective context. The transition system is shown in Figure 2.11. There are six transitions in this figure having guards where $i$ is the number of processes currently in the barrier: (1) Process $i < N$ enters the barrier, (2) Process $i = N$ enters the barrier, (3) Process $i < N$ enters the barrier, (4) Process $i = N$ enters the barrier, (5) Process $i > 1$ exits the barrier, and (6) Process $i = 1$ exits the barrier. The actions are to change the state of the collective context and either add or remove the process’s PID from the participating set as the transition requires. All barrier entrance transitions add the process’s PID; exit transitions remove the process’s PID.

![Figure 2.11. The state machine for the barrier synchronization protocol.](image-url)
2.12 The Initial State of the Model

The initial state of the model is required in order to facilitate model checking with respect to the operations as described. Each request sequence is initialized having a single request at index zero, the buffer size for each process is set to zero, each process’s message buffer is initialized to represent no available buffering, each process is marked as uninitialized, and each process receives a copy of the default communicator MPI_COMM_WORLD.

The rank order is also decided at the initial state. This is the function that maps process identifiers onto ranks in N. If the order is significant, the user must specify this in the configuration file. In this case all possible permutations are tried when the model is examined by the model checker. Otherwise, a single permutation is chosen arbitrarily.

2.13 Issues Raised by Modeling

While creating the model some specific issues that had not been discussed in the MPI natural language version of the standard became apparent. The following descriptions are helpful in understanding the following issues. MPI_Probe takes a process rank $j$ and some additional message envelope information and becomes enabled when there is a matching request posted on process $j$. MPI_Cancel takes a request handle as an argument and attempts to cancel the corresponding communication. The standard says the message may still complete, and it is up to the user to program appropriately. A third operation MPI_Rsend (and variants) requires the matching receive operation to have been previously posted, barring which the operation is in error. In this context, here are some specific issues that have been identified:

- There are numerous ways that MPI_Probe and MPI_Cancel can interact, resulting in an undefined system state. In particular, any time a message is probed successfully, it is not specified whether it is still possible for the message to be canceled or if the message must be delivered once successfully probed.
• **MPI_Cancel** also creates an undefined system state when used with ready mode send (**MPI_Irsend**). Consider the following execution trace:
P0: h=MPI_Irecv 1; P1: MPI_Irsend 0; P0: MPI_Cancel h; ... If the ready send is successful, can the receive still be canceled?

• Continuing with **MPI_Cancel**, what happens if the null request is canceled?

• The MPI system allows the user to specify a buffer for outgoing messages. To ensure that all buffered messages have been sent, the user must call **MPI_Buffer_detach**. What is the state of the system when no buffer has been specified and **MPI_Buffer_detach** is called?

• The resultant state of several operations is unspecified when used with the constant rank **MPI_PROC_NULL**. For example: what happens when a **MPI_Probe** operation is executed with **MPI_PROC_NULL**?

• The resultant state of the status variables in several scenarios is unspecified. For example: what are the values of the status variable fields when **MPI_Probe** is called with **MPI_PROC_NULL** as the source?

### 2.14 Related Work

Each formal specification framework solves modeling and analysis issues specific to the object being described. In this case, it was not initially clear how to handle the daunting complexity of MPI nor how to handle its modeling, given that there has only been very limited effort in terms of formal characterization of MPI. The architecture of the model that incorporates numerous solutions is presented in this chapter.

In [27], Georgelin and Pierre specify some of the MPI functions in LOTOS [20]. Their model includes **MPI_Send**, **MPI_Ssend**, **MPI_Bsend**, **MPI_Rsend**, **MPI_Scatter**, **MPI_Gather**, and **MPI_Broadcast**. Although they mention the immediate mode versions of the above, no models of the immediate mode operations are shown, and no mention is made of persistent requests and many other operations.
In [77], Siegel and Avrunin describe a finite state model of a limited number of MPI point to point operations. This finite state model is embedded in the SPIN model checker [36]. In [79], the authors support a limited partial order reduction method — one that handles wild-card communications in a restricted manner. The authors make assumptions about the future of the execution trace to ensure that the reduction is sound. However such assumptions are impractical as it is possible for an additional process to match a wildcard receive at some time in the future, regardless of the skew in execution synchronization displayed in the current state.

In [76], additional ‘nonblocking’ MPI primitives are modeled in SPIN. Here the authors report that these primitives have been implemented, and give a finite state machine describing the implementation, however there is no high level mathematical statement about the semantics such as the TLA+ MPI model.

Past efforts in this area are described in [6, 65, 66, 72]. Implementations have been attempted using Promela, Zing, and +CAL. Each of these attempts found the expressivity of the modeling language wanting as the implementation details (such as the use of queues to enforce FIFO message delivery) must become explicit.

None of these efforts approach the number of MPI functions handled in this model, or have the same style of high level specifications, TLA+ being much closer to mathematical logic than finite-state SPIN or LOTOS models.

2.15 Summary

This chapter has described a formalization of a subset of operations described in the MPI 1.1 standard using the temporal logic of actions, TLA+. The full model is included in Appendix A. The abstractions made to create the model have been discussed, assertions and invariants that the model imposes, and how the individual point to point operations contained in the model can be composed to formulate the various composite point to point operations prescribed by the MPI standard. The design of this MPI model is sufficiently general to allow for additions such as floating point numbers, collective operations, and virtual topologies, in addition to handling the 35 point to point operations in a uniform way.
The TLA+ model of MPI is an important contribution because it precisely specifies and disambiguates the semantic interaction of the many different point to point operations. The specification is cross referenced with the natural language standard, such that the page and line number in the standard can be corroborated for many of the logic clauses in the model, something no other model of MPI has offered. This specification can also serve as a design guide for customized model checkers (as shown in Chapters 3 and 5). In addition the specification can be checked against program models that are extracted using the framework of Chapter 4.
CHAPTER 3

DYNAMIC PARTIAL-ORDER REDUCTION FOR MPI

In the previous chapter a high level, mathematically precise, model of the communication semantics for a large subset of the operations described in the MPI 1.1 standard is presented. The semantics as they are expressed in TLA+ can be used for small litmus test based model checking examination. This works well unless there are more than a few concurrent processes examined by the model checker.

To see why examining a model with more than a few processes can be a problem for a model checker such as TLC, consider an example of two processes, each having three states and two transitions. Under an interleaving semantics, when the full state space is explored, six states will be generated. Now, if the number of processes is increased to three, $3^3 = 27$ states will be generated. In general the number of states of a single process ($p$) raised to the number of processes $N$ (i.e., $p^N$) is the upper bound on the size of the reachable state space of a concurrent system.

The most effective approach to mitigate this state-space explosion is partial-order reduction. Partial-order reduction works by delaying the actions of some processes in a way that is not observable by the property being checked. These scheduling delays cause the model checker to visit only a subset of the full reachable state space. A sound partial-order reduction guarantees that if there is a property violation in the full reachable state space, that violation will be discovered by the model checker while enumerating the selected subset of the reachable state space.

Each of the transition traces that are visited by the model checker represent many possible temporal permutations of transitions, often referred to as execution interleaving orders. Most partial-order reduction algorithms use some notion of independence to determine when it is safe to allow a subset of the enabled processes
at a given state to execute. This independence is computed based on the semantics of a transition — what variables the transition touches, whether the transition performs reads or writes with respect to those variables, whether the variables are property observable, etc. Thus for a library based communication system, the independence characteristics of the library operations must be argued based on the properties being preserved by the reduction and either the library’s formal semantics or the library’s implementation.

In Chapter 2 a formal semantic model of MPI is presented. This model includes all point to point operations and many supporting operations described in the MPI 1.1 standard as a collection of TLA+ model transitions. TLC, the model checker for specifications written in TLA+, is extremely general and does not compute independence relations for any transition systems, nor does it use any partial-order reduction algorithms while model checking. Therefore, models that require more than a few processes, cause the TLC model checker to exceed the memory limits set, or to run beyond any reasonable time bound, or both.

Prior work attempts to effect a partial-order reduction by strengthening the guards of the TLA+ model. A scheduler transition is imposed on the set of processes that would choose an individual process to execute only under a set of given constraints (described in [67]). In reasoning about this approach, it became clear that a primary constraint to determine independence was when an MPI_Wait corresponding to an MPI_Irecv matches with an MPI_Send (or variant) of another process. Information regarding which messages may result in a communication is available to the model checker while model checking, only when the maximal trace has been computed. This implies the need for more information than is available in a single state to determine the independence of the enabled transitions at that state — needing to be able to see into the future!

Dynamic partial-order reduction [24] is a way to use information about the future of an execution to determine when two processes will in fact communicate. This chapter attempts to answer these two needs: (i) What are the independence characteristics of a subset of the MPI point-to-point operations? and (ii) How can
information about the future (i.e., dynamic partial-order reduction) be applied to model checking MPI-based program models?

The chapter proceeds as follows: Section 3.1 gives a number of examples to motivate the need for specialized partial-order reduction for MPI. Section 3.2 gives a simple modeling language that includes MPI based primitives and a set of semantic definitions that create a closed world. In this section a number of independence theorems are stated regarding the individual transition types and proofs of these theorems are presented. Section 3.3 presents a dynamic partial-order reduction algorithm that uses the independence proofs given in Section 3.2. This algorithm is tailored specifically to the properties of interest and to the MPI semantics. In this section a proof that the dynamic partial-order reduction algorithm preserves local assertions, deadlocks, and the presence of cycles is presented. Section 3.4 describes an example of the effectiveness of this approach, while a detailed example is explained in Section 3.5. Section 3.6 gives an overview of related partial-order reduction literature. Section 3.7 summarizes this chapter.

3.1 Motivating Examples

This section provides a few examples of MPI programs, partly to give the reader a basic understanding of MPI, partly for illustration later, and partly to denote issues which this approach cannot yet handle. The subset of MPI operations modeled in this chapter includes MPI_Issend, MPI_Irecv, MPI_Wait, MPI_Test, and MPI_Barrier; wildcard receive operations are allowed. It is assumed that the model is provided in SPMD style, where each process in the computation executes the same program image. For brevity, the prefix “MPI_” will be omitted in the remainder of this chapter and use the regular fonts except when making direct reference to some pseudo-code, e.g., MPI_Wait will be referred to as Wait etc.

The $N$ processes in an MPI distributed computation can be differentiated based on the unique process rank (integers in the closed interval $[0, N - 1]$) that the MPI runtime system assigns to each MPI process by calling Comm_rank. Messages are addressed by process rank (“rank”) or through wildcards (denoted by *) that stand
for \texttt{ANY\_SOURCE}. Consider the simple communication pattern where all processes send a message to the root process (rank 0) implemented in Figure 3.1. Here, \texttt{Issend 0} means “send to 0” while \texttt{Irecv i} means “receive from i.” This code will deadlock as it contains a mismatched communication: waiting (through the \texttt{wait h} command) for an Irecv with source rank 0 to complete before executing the Irecv operations for other ranks.

Further considering the example in Figure 3.1, which actions in this program are independent? It turns out that the Issend operations from line 2 are dependent only on the Wait corresponding to the Irecv operations on line 7 when \texttt{i} in the Irecv on line 6 is equal to the rank of the sending process! Similarly the Wait on line 3 is dependent only on the Irecv of line 6 when \texttt{i} is equal to the rank of the sending process. All other program actions are independent. All this information is a direct consequence of the formal semantics. Without a formal semantics, finding such dependencies becomes difficult.

Figure 3.2 shows a second example where a process can receive from multiple potential sources in any order. Suppose the pseudo-code for this example is instantiated with four (4) processes. Here the developer expected the process with \texttt{rank == 0} to receive the last message from the process with \texttt{rank == 3}. However unless this is guaranteed using some other synchronization mechanism, it will be possible to violate this assertion. This example underscores the importance of

```
1 if(rank != 0){
2   h = Issend 0 (addrof x)
3   Wait h
4 } else {
5   for(int i = 0; i < N; ++i){
6     h = Irecv i (addrof x)
7     Wait h
8   }
9 }
```

\textbf{Figure 3.1.} A program pseudo-code that contains a deadlock caused by a mismatched handshake.
model checking based verification of MPI programs, as opposed to random-testing which can miss such bugs.

The next example (Figure 3.3) demonstrates the need to use dynamic information in the computation of dependencies. Let this program be instantiated for \( N = 6 \) processes. This program causes ranks 1 and 2 to send a message to rank 3; ranks 4 and 5 also send to rank 0. The messages can be received in either order — meaning that the value in \( x \) on rank 0, after the "Wait \( h \);" executes, could be either from rank 5 or rank 6. Moreover, since the Test operation is used on line 4 (as opposed to a Wait) it is possible that the first receive does not complete in this part of the program. Although dependencies exist, the dynamic partial-order reduction algorithm naturally forms clusters of independence e.g., as in [7]; however, unlike in their work, the clusters will not be statically and a priori determined.

### 3.2 Language Execution and Communication Semantics

In this section, the execution semantics for a simple goto-based program modeling language for MPI called MPIC is presented. MPIC is a simplification of
1 if(rank%3==0){
2   h = Irecv * (addrof x);
3   i = Irecv * (addrof y);
4   flag = Test h;
5   Wait i;
6 } else {
7   if((rank+2)%3==0){
8     h = Issend (rank+2)%rank (addrof x);
9   } else {
10      h = Issend (rank+1)%rank (addrof x);
11   }
12   Wait h;
13 }

Figure 3.3. A program pseudo-code fragment illustrating the role of dynamic information in partial-order reduction for MPI.

the much more detailed semantics of MPI is discussed in Chapter 2 and shown in Appendix A. The grammar for this language is shown in Figure 3.4. The C convention in modeling Boolean operations is followed, using Integers. Nonzero values are true; zero is false. The standard complement of arithmetic, logical, and program operators are assumed for \textit{binaryop}, and \textit{unaryop}. For communication a representative operation is chosen from each of the point-to-point operation groups.

The communication operations were chosen for their applicability in deterministic optimization. A program that implements message coalescing and queuing for enhanced communication efficiency would use Issend and Irecv to indicate to the MPI subsystem that additional buffering is unnecessary — thereby increasing performance. \textit{Wait} and \textit{Test} both complete communications. \textit{Wait} blocks until the communication completes, whereas \textit{Test} is always enabled and returns in a flag whether the communication has completed. \textit{Barrier} is used to conservatively represent all of the collective communication operations of MPI — any of which may have a synchronizing effect (including the actual Barrier function supported by MPI that is always synchronizing).

\textbf{Definition 1 (Request)} A request is a five-tuple \textit{(source, destination, address,
const ::= int
var ::= intvar
intexp ::= const
| var
| intexp binaryop intexp
| unaryop intexp
labexp ::= label
| 'if' intexp 'then' label 'else' label
exp ::= intexp
| ('exp')
localinstr ::= var '=' exp
| 'goto' labexp
| 'assert' intexp
comminstr ::= int '=' 'issend' intexp
| int '=' 'irecv' intexp
| bool '=' 'test' intvar
| 'wait' intvar
| 'barrier_init'
| 'barrier_wait'
instr ::= localinstr ';' instr
| localinstr
| comminstr ';' instr
| comminstr
| label ';' instr
prog ::= (instr)+

Figure 3.4. The Grammar of the MPIC modeling language.

type, completed) where

source ∈ N
destination ∈ N
address ∈ N
type ∈ {send, recv}
completed ∈ B

Definition 2 (Collective Context) The collective context is a pair

c : (collective, participants) where

collective ∈ {vacant, in, out}
participants ∈ P({i : i ∈ 0..N − 1})

representing the state of the collective operation and the processes that are currently participating.
Definition 3 (Process) A process is a pair \((l, g)\) where \(l\) is a function \(l : \mathbb{N} \rightarrow \mathbb{N}\) representing the local memory of the process (i.e., the variables used in the program) and \(g\) is a function \(g : \mathbb{N} \rightarrow \text{Request}\).

Definition 4 (State) A state is a pair \((c, p)\) where \(c\) is a collective context and \(p\) is a function \(p : \mathbb{N} \rightarrow \text{Process}\).

A state is a tuple \((c, p)\) where \(c\) is the collective context and \(p\) is a function mapping process identifiers (PID$s in \mathbb{N}$) onto the state of each process. Processes are individually modeled by a local store \(l\) and global store \(g\). Each local store \(l_i\) of process \(i\) is a function from addresses in \(\mathbb{N}\) onto values also in \(\mathbb{N}\). The global store can only be accesses via MPI operations. Each global store \(g_i\) is a function from PID$s to a set of request tuples. A request is a five-tuple \(\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathbb{B}\) indicating the source, destination, memory address to be read or written, message type (being either send or receive), and an indication whether the message has been completed. The state of collective operations such as Barrier are recorded in the collective context \(c\). The parameter \(N\) is constant in this system and represents the number of processes participating in the distributed computation.

Figure 3.5 provides four helper functions that are used in the remainder of the presentation. Communications between processes are determined using meta-data provided in the request. When two requests can form a communication, they \textit{Match}. The predicate \textit{Match} evaluates to true when two message requests, one sending and the other receiving, can be combined to create a communication. The functions Completed, Type, and Address, provide access to the corresponding fields of the request tuple.

Each of the rules in Figures 3.6, 3.7, 3.8, and 3.9 manipulate the state tuple. Operations with multiple rules model the disjoint nature of the transition type. The Barrier operation is modeled using six rules implementing two transition types: an entrance and an exit. The reachable state space, viewed as a unary predicate \(\Sigma\), is inductively defined by the execution of these rules parameterized over the transition relation of a given program and some fixed number of processes \(N\). The transition
Match \(( (\text{src}_1, \text{dest}_1, _, \text{type}_1, \text{completed}_1), (\text{src}_2, \text{dest}_2, _, \text{type}_2, \text{completed}_2) ) = \\
\land \text{dest}_1 = \text{dest}_2 \\
\land \text{completed}_1 = \text{false} \\
\land \text{completed}_2 = \text{false} \\
\land \lor \text{src}_1 = \text{src}_2 \\
\lor \land \text{type}_1 = \text{recv} \\
\land \text{src}_1 = * \\
\lor \land \text{type}_2 = \text{recv} \\
\land \text{src}_2 = * \\
\land \lor \land \text{type}_1 = \text{send} \\
\land \text{type}_2 = \text{recv} \\
\lor \land \text{type}_1 = \text{recv} \\
\land \text{type}_2 = \text{send} \)

Completed \((_, _, _, _, c) = c \)

Type \((_, _, t, _) = t \)

Address \((_, _, a, _, _) = a \)

**Figure 3.5.** Helper functions used in the communication semantics.

relation is defined using two functions. The \textit{proc} function returns the abstract syntax tree node for a given \textit{pc} value. Sequential control flow in the model being examined is represented by the \textit{next} function. The \textit{pc} \& \epsilon \textit{N} is held in the local store of each process.

Although suppressed for simplicity in this presentation, data is transmitted between processes in the Wait (2) and Test (2) rules.

Again consider the example of Figure 3.1. This program uses Assignment, Goto, Issend, Irecv, and Wait. It is assumed that a suitable evaluator \( E \) exists \((E : \text{exp} \rightarrow \textit{N})\) for expressions in the language and that control flow is simplified into conditional goto statements. The rule for Assignment says that if there is a state \((c, p)\) such that \textit{proc} maps to an assignment for the current \textit{pc} of process \(i\), in the next state the system updates the value of the \textit{pc} with \textit{next}(\textit{pc}), and assigns the evaluated
Assignment:
\[
\begin{align*}
\sum(c, p) \land p(i) = (l, g) \land proc(l(pc)) &= (assign \ x \ e) \\
\sum(c, p[i \mapsto (l[pc \mapsto next(pc)], E[(addrof x), p(i)] \mapsto E[e, p(i)]], g))
\end{align*}
\]

Goto:
\[
\begin{align*}
\sum(c, p) \land p(i) = (l, g) \land proc(l(pc)) &= (goto \ e) \\
\sum(c, p[i \mapsto (l[pc \mapsto E[e, p(i)]], g)])
\end{align*}
\]

Assert:
\[
\begin{align*}
\sum(c, p) \land p(i) = (l, g) \land proc(l(pc)) &= (assert \ e) \land E[e, p(i)] = true \\
\sum(c, p[i \mapsto (l[pc \mapsto next(l(pc)]), g)])
\end{align*}
\]

Issend:
\[
\begin{align*}
\sum(c, p) \land p(i) = (l, g) \land proc(l(pc)) &= (issend \ x \ dest \ addr) \\
\sum(c, p[i \mapsto (l[pc \mapsto next(l(pc)]), E[(addrof x), p(i)] \mapsto (\text{Dom}(g) + 1), g[(\text{Dom}(g) + 1) \mapsto (i, E[dest, p(i)], E[addr, p(i)], send, false)])])
\end{align*}
\]

Irecv:
\[
\begin{align*}
\sum(c, p) \land p(i) = (l, g) \land proc(l(pc)) &= (irecv \ x \ src \ addr) \\
\sum(c, p[i \mapsto (l[pc \mapsto next(l(pc)]), E[(addrof x), p(i)] \mapsto (\text{Dom}(g) + 1), g[(\text{Dom}(g) + 1) \mapsto (E[src, p(i)], i, E[addr, p(i)], recv, false)])])
\end{align*}
\]

**Figure 3.6.** The semantic definitions of Assignment, Assert, Goto, Issend, and Irecv.

value of expression e in the current state to the memory location referenced by x. Goto is very similar except the expression evaluated is assigned to the pc in the next state. Issend and Irecv are again similar with the exception that x contains a numeric handle to the request that is created by the execution of this operation.

Some detail about requests is necessary before elaborating on the semantics of Wait. The natural language MPI standard [82] uses the term request extensively without giving a formal definition to it. Send and receive operations, regardless of their type (only Issend and Irecv are discussed in this chapter), all produce or activate requests; Wait and Test operations consume or deactivate and possibly deallocate requests depending on the flavor of operation and request. In this chapter a request is defined to be a five-tuple that contains the rank of the sender and
Barrier_init (1):

\[\Sigma(((vacant, \emptyset), p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{barrier_init}) \land \Sigma((in, \{i\}), p[i \mapsto (l[pc \mapsto next(l(pc))], g)])\]

Barrier_init (2):

\[\Sigma(((in, s), p) \land p(i) = (l, g) \land i \notin s \land proc(l(pc)) = (\text{barrier_init}) \land \Sigma((in \cup \{i\}, p[i \mapsto (l[pc \mapsto next(l(pc))], g)])\]

Barrier_wait (1):

\[\Sigma(((in, s), p) \land p(i) = (l, g) \land s = \{i : i \in 0..(N - 1) \land s \neq \{i\} \land proc(l(pc)) = (\text{barrier_wait}) \land \Sigma((out, s \setminus i), p[i \mapsto (l[pc \mapsto next(l(pc))], g)])\]

Barrier_wait (2):

\[\Sigma(((in, s), p) \land p(i) = (l, g) \land s = \{i : i \in 0..(N - 1) \land s = \{i\} \land proc(l(pc)) = (\text{barrier_wait}) \land \Sigma((vacant, \emptyset), p[i \mapsto (l[pc \mapsto next(l(pc))], g)])\]

Barrier_wait (3):

\[\Sigma((out, s), p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{barrier_wait}) \land i \in s \land s \setminus i \neq \emptyset \land \Sigma((out, s \setminus i), p[i \mapsto (l[pc \mapsto next(l(pc))], g)])\]

Barrier_wait (4):

\[\Sigma((out, s), p) \land p(i) = (l, g) \land proc(l(pc)) = (\text{barrier_wait}) \land i \in s \land s \setminus i = \emptyset \land \Sigma((vacant, \emptyset), p[i \mapsto (l[pc \mapsto next(l(pc))], g)])\]

Figure 3.7. The semantic definitions of Barrier_init and Barrier_wait.

receiver, the address of memory to be read or written, the type of message being requested (either send or receive), and whether or not this particular request has been completed. Moreover, deallocation and deactivation is ignored and focus only on the producer/consumer relationship that exists between Issend, Irecv, Wait, and Test in this restricted context. A model that admits more of these operations (such as Recv_init and Start) would have to take into account the additional complexity (cf. the TLA+ model of MPI in Appendix A).
Figure 3.8. The semantic definition of Wait.

A process completes the communication initiated by an Issend or Irecv by executing a Wait or Test. The Wait operation is paired with an Issend or Irecv on a particular process via a request handle returned by the Issend or Irecv. The execution semantics are similar to Assignment as it requires that some process \( i \) be positioned to execute a Wait operation in the current state. The additional restriction for Wait (1) is that either the request handle passed to the Wait evaluates to zero (\( E[e,p(i)] = 0 \)), a special value outside \( \text{Dom}(g) \) used to represent REQUEST_NULL, or if the handle is valid then the request has been completed by the Wait or Test of another process. In this case the \( pc \) of the process is updated to next(l(pc)) and the handle is set to 0. The additional restriction for Wait (2) is that the request handle evaluate to a value in \( \text{Dom}(g_j) \) and there must be some other request on process \( j \) that will Match the request \( g(E[e,p(i)]) \). The final clause, i.e. \( m < k \Rightarrow \neg Match(g_j(m),g_i(E[e,p(i)])) \), in connection with the deterministic structure of a single process, enforces the program order matching requirement for requests imposed by MPI. In this case the \( pc \) and handle are updated as before. In addition, the global store is modified to reflect that the communication has completed. This is shown in the semantics as \( g_i(E[e,p(i)])[false/true] \) which is the request at index \( E[e,p(i)] \) with every occurrence of false replaced with true.
Test (1):

\[
\Sigma(c, p) \land p(i) = (l_i, g_i) \land \text{proc}(l_i(pc)) = \text{(test } e_1\text{ } e_2) \land \\
(E[e_1, p(i)] = 0 \lor (E[e_1, p(i)] \in \text{Dom}(g) \land \text{Completed}(g(E[e_1, p(i)])))
\]

\[
\Sigma(c, p[i] \mapsto (l_i[pc \mapsto \text{next}(l_i(pc)), E[(\text{addrof} e_1), p(i)] \mapsto 0, \\
E[(\text{addrof} e_2), p(i)] \mapsto \text{true}], g))
\]

Test (2):

\[
\Sigma(c, p) \land p(i) = (l_i, g_i) \land \text{proc}(l_i(pc)) = \text{(test } e_1\text{ } e_2) \land \\
E[e_1, p(i)] \in \text{Dom}(g_i) \land \text{Match}(g_j(k), g_i(E[e_1, p(i)])) \land \\
m < k \Rightarrow \neg\text{Match}(g_j(m), g_i(E[e, p(i)]))
\]

\[
\Sigma(c, p[i] \mapsto (l_i[pc \mapsto \text{next}(l_i(pc)), E[e_1, p(i)] \mapsto 0, E[e_2, p(i)] \mapsto \text{true}], \\
g_i[E[e_1, p_i]] \mapsto g_i(E[e_1, p_i])[\text{false/true}]), j \mapsto (l_j, g_j[k \mapsto g_j(k)[\text{false/true]}]))
\]

Test (3):

\[
\Sigma(c, p) \land p(i) = (l_i, g_i) \land \text{proc}(l_i(pc)) = \text{(test } e_1\text{ } e_2) \land \\
E[e_1, p(i)] \in \text{Dom}(g_i) \land \neg\text{Completed}(g_i(E[e_1, p(i)])) \land \\
p(j) = (l_j, g_j) \land \neg\text{Match}(g_j(k), g_i(E[e_1, p(i)]))
\]

\[
\Sigma(c, p[i] \mapsto (l_i[pc \mapsto \text{next}(l_i(pc)), E[e_2, p(i)] \mapsto \text{false}], g))
\]

**Figure 3.9.** The semantic definition of Test.

The data are also moved from sender to receiver in this step although tacit in this presentation.

This model allows for corner cases such as calling Wait on the same handle variable twice — the second call should not block. In addition, the value 0 is used to model the `REQUEST_NULL` constant of the MPI standard. The only ways a request handle can have the `REQUEST_NULL` value is by either explicit use in the source program or by executing a Wait transition with a valid handle.

The rules for Test are very similar to those of Wait with the addition of a third rule that makes Test always enabled. In this case (Test (3)) the local operation has not completed and there is no request posted in the global stores of any process that could enable Test (2).

The model of this chapter is a significantly simplified version of the model from Chapter 2 intended to facilitate experiments with partial-order reduction. In that
respect, this model is superior in that it trades away the rich variety of operations for a set of semantics that is small enough to reason about by hand.

### 3.2.1 Assumptions and Properties of Interest

Some well-formedness assumptions are made of MPI program models that can be handled by the DPOR algorithm. It is assumed that a process does not access the buffer passed to an Issend or Irecv until after the Wait or Test returns and if Test is used, the flag returned by Test is true. More formally, for any process \( i \), if \( a \) is an address such that \( \exists r : (\text{Address}(g_i(r)) = a) \land \neg\text{Completed}(g_i(r)) \) then it assumed that no other action of process \( i \) will read or write \( l_i(a) \). This assumption is necessary for correctness according to the MPI standard [82]. An example that violates this assumption is: \( h = \text{irecv 3 (addr0f h)} \), where \( h \) is assigned a value before a Wait or Test on \( h \) has been performed. A second assumption is that programs cannot explicitly reference the program counter of any process, e.g., statements of the form \( \text{pc} = \text{exp} \) and \( v = \text{irecv 2 (addr0f pc)} \) are a violation of this assumption. It possible to check these assumptions, with modest or no over-approximations, using existing static analysis methods. A third assumption is that along every program path, each Issend and Irecv have a corresponding Wait or Test that completes the communication.

Only certain properties are preserved by the reduction algorithm. In particular, under this execution semantics, the reduction algorithm preserves (i) deadlocks, (ii) the presence of cycles\(^1\), and (iii) local invariance assertions, more formally, assertions on \( \text{Dom}(l_i) \) for each process \( i \) that are invariant under stuttering.

### 3.2.2 Independence Characteristics of MPI Operations

The independence properties of the transition semantics presented in Figures 3.6, 3.7, 3.8, and 3.9 is now presented. For completeness the definition of independence from [30] is restated here. For any state \( \sigma \in \Sigma \), the set of transitions enabled in \( \sigma \), \( \text{enabled}(\sigma) \) is defined as \( \text{enabled}(\sigma) = \{ t_i \mid t_i(\sigma) \in \Sigma \} \). A transition

---

\(^1\)The MPI standard requires all processes to eventually call Finalize, after which there can be no more communication via the MPI library. Therefore, all cycles are considered errors.
$t_i$ of process $i$ and a transition $t_j$ of process $j$ where $i \neq j$ are independent (i.e., $I(t_i,t_j)$) if
\[\forall \sigma \in \Sigma : t_i,t_j \in \text{enabled}(\sigma) \Rightarrow (t_i \in \text{enabled}(t_j(\sigma)) \land t_i(t_j(\sigma)) = t_j(t_i(\sigma))).\]

Two transitions $t_i$ and $t_j$ are dependent if $\neg I(t_i,t_j)$.

With the transition semantics as defined, a number of theorems will be stated and proven. For brevity only proof sketches are provided, keeping the details for Section 3.2.3.

**Theorem 1**  If $t_i$ is an Assignment, Goto, Assert, or Barrier transition for a given process $i$ and $t_j$ is any transition of process $j$ where $i \neq j$ then $I(t_i,t_j)$.

**Proof.** (Sketch) To prove this theorem for Assignment, Goto, and Assert it is sufficient to note that disjoint memory spaces are read and written. For each of the Barrier transitions, the key observation is that the execution of other processes cannot disable the enabled transition into or out of a Barrier. In both cases independence is with respect to observable behavior.

Corresponding theorems for Issend, Irecv, Wait, and Test require some additional proof machinery. Figure 3.10 gives the definition for the predicate Complete and pseudo-code for the Index operation. The predicate Complete evaluates to true for two transitions when they could be used to cause a communication between two processes or when they can cause two processes to not communicate. In the pseudo-code, $\text{post}(t)$ is the state generated by executing transition $t$, $\text{proc}(t)$ is the process that executed $t$. The Index operation takes a transition $t$ as an argument and returns the handle of the request referenced by $t$.

**Theorem 2**  If $t_i$ is an Issend, Irecv, Wait, or Test of process $i$ and $t_j$ is any transition of another process $j$ where $i \neq j$ and $\neg \text{Complete}(t_i,t_j)$ then $I(t_i,t_j)$.

**Proof.** (Sketch) The key observation is that the transitions involved read from and write to disjoint memory spaces.

Intuitively, Complete indicates when two transitions could result in, or interfere with a communication under the right execution interleaving order. This theorem
Index(t) ≡
IF t is a Wait or Test
THEN
return the evaluated handle argument
ELSE
IF t is an Issend or Irecv
return the value written into the handle address
ELSE
return the null request value

Complete(α, β) ≡
Let (c₁, p₁) = post(α) in
Let (l₁, g₁) = p₁(proc(α)) in
Let (c₂, p₂) = post(β) in
Let (l₂, g₂) = p₂(proc(β)) in
Let i = Index(α) in
Let j = Index(β) in
IF ¬(i = null)∧¬(j = null)
THEN
Match(g₁(i), g₂(j))
ELSE
false

Figure 3.10. Definition of the predicate Complete.

says that transitions that cannot result in a communication are independent with respect to the observable behavior.

The consequence of the above two theorems is that all of the program’s non-communication actions are independent and many of the communication actions are also independent. Moreover, a simple predicate (i.e., Complete) is given that can be evaluated during model checking to determine whether two MPI communication operations are in fact dependent. Using this predicate, the questions from Section 3.1 pertaining to the independence of actions in Example 3.1 can be approximated.

3.2.3 Proof of the Independence Theorems for MPI Operations
This section provides the above Lemmas and their proofs in greater detail. For each of the proofs to follow it is assumed the requirements of Section 3.2.1 hold.
Lemma 1 If $t_i$ is an Assignment, Goto, or Assert transition for a given process $i$, and $t_j$ is any transition of process $j$, where $i \neq j$, then $I(t_i, t_j)$.

Proof. To show the enabledness condition, suppose $t_i$ is an Assignment, Goto, or Assert transition of process $i$, $t_j$ is any transition of process $j$ where $i \neq j$ and $t_i, t_j \in enabled(\sigma)$ for some $\sigma \in \Sigma$. Now suppose that $t_i$ is disabled in $t_j(\sigma)$. Then the action of process $j$ changed the local store $l_i$ of process $i$ (as the enabling condition for $t_i$ is only dependent on $l_i$). Since none of the semantic rules allows such a modification, there is a contradiction. Now suppose that $t_j$ is disabled in $t_i(\sigma)$. Only $t_i$ is able to write the local store of process $i$. Therefore it is not possible to disable $t_j$.

Now to show the commutativity property, again suppose that $t_i$ is a transition, as before, of process $i$ enabled at $\sigma$ and $t_j$ is a transition of process $j$ where $t_i, t_j \in enabled(\sigma)$. Since $t_i \in enabled(t_j(\sigma)) \land \Sigma t_j(\sigma)$ and $t_j \in enabled(t_i(\sigma)) \land \Sigma t_i(\sigma)$, clearly there exists some $\sigma_1$ and $\sigma_2$, such that $\sigma_1 = t_i(t_j(\sigma))$ and $\sigma_2 = t_j(t_i(\sigma))$. Therefore, $\sigma_1 = \sigma_2$ as the modified stores are disjoint. 

Lemma 2 If $t_i$ is a Barrier transition for a given process $i$ and $t_j$ is any transition of process $j$ where $i \neq j$ then $I(t_i, t_j)$.

Proof. To show the enabledness condition, suppose $t_i$ is a Barrier transition of process $i$, $t_j$ is any transition of another process, $i \neq j$, and $t_i, t_j \in enabled(\sigma)$ for some $\sigma \in \Sigma$. Now suppose for contradiction that executing $t_j$ causes $t_i$ to become disabled. The enabledness conditions of $t_i$ are dependent only upon actions of process $i$ and the state of the collective context. The collective context changes only in response to all processes entering or exiting the barrier. Therefore, $t_i$ was disabled by an action of process $i$, a contradiction. Now suppose that $t_j$ is disabled in $t_i(\sigma)$. Since $t_i$ modifies only the collective context of $\sigma$ and the $l_i(pc)$ of process $i$ then $t_j$ must be a Barrier operation. If $t_j$ is a Barrier init that is enabled in $\sigma$ and disabled in $t_i(\sigma)$ then $t_i$ must have added $j$ to the collective context — a contradiction. If $t_j$ is a Barrier wait that is enabled in $\sigma$ and disabled in $t_i(\sigma)$ then
Now to show the commutativity condition, suppose $t_i$ and $t_j$ are Barrier transitions of processes $i$ and $j$ respectively (all other transitions commute trivially). Further suppose $t_i, t_j \in enabled(\sigma)$ for some state $\sigma$. It can concluded that $t_i$ and $t_j$ are both the same type of Barrier transition (either init or wait). In this case, note that the set $s$ is unordered and the change in the collective context state would be the same after executing in either order. Therefore $t_i(t_j(\sigma)) = t_j(t_i(\sigma))$. \hfill \blacksquare

One interesting consequence of Lemma 2 is that it is unnecessary to examine more than one execution interleaving order of processes entering and exiting barrier operations. This can greatly reduce the cost of analysis when barriers are heavily used by a program model to force lock-step execution.

**Definition 5** Transitions $t_1$ is said to complete $t_2$ if $Complete(t_1, t_2)$ as defined in Figure 3.10 evaluates to true.

**Lemma 3** If $t_i$ is an Issend or an Irecv of process $i$ and $t_j$ is any transition of another process $j$ where $i \neq j$ and $\neg Complete(t_i, t_j)$, then $I(t_i, t_j)$.

**Proof.** To show enabledness, suppose $t_i$ is an Issend or Irecv of process $i$ and $t_j$ is any transition of process $j$, $i \neq j$, where $t_i, t_j \in enabled(\sigma)$ for some $\sigma \in \Sigma$. Suppose to the contrary that $t_i \notin enabled(t_j(\sigma))$. Then $t_j$ must have written $l_i(pc)$. It has been assumed that this is not possible. Now suppose that $t_j \notin enabled(t_i(\sigma))$. Since $t_i$ only writes the local and global stores of process $i$, it is not possible to disable any transition of another process and in particular $t_j$.

To show commutativity, since $\neg Complete(t_i, t_j)$, regardless of the transition type of $t_j$, under the well-formedness assumptions, $t_i$ and $t_j$ write and read disjoint memory locations. Therefore $t_i(t_j(\sigma)) = t_j(t_i(\sigma))$. \hfill \blacksquare

**Lemma 4** If $t_i$ is a Wait of process $i$ and $t_j$ is any transition of another process where $i \neq j$ and $\neg Complete(t_i, t_j)$ then $I(t_i, t_j)$.
Proof. To show enabledness, suppose $t_i$ is a Wait of process $i$ and $t_j$ is any transition of process $j$ where $t_i, t_j \in enabled(\sigma)$ for some $\sigma \in \Sigma$. Suppose to the contrary that $t_i \notin enabled(t_j(\sigma))$. Then $t_j$ must have modified some of the enabling conditions of the Wait. It is the case that $t_j$ could have written to the local or global store of process $i$ only if it were completing the message between processes $j$ and $i$ (implying $t_j$ is a Wait or Test transition). However, it has been assumed that $\neg Complete(t_i, t_j)$, meaning that if $t_j$ were a Wait or Test, the corresponding request would not Match the request of $t_i$ and therefore would reference disjoint memory space under the well-formedness assumptions. Since no other transition could change the enabling condition of $t_i$, $t_i \in enabled(t_j(\sigma))$. Now suppose $t_j \notin enabled(t_i(\sigma))$. Then $t_i$ must have written $l_j(pc)$ or $t_j$ was a Wait on a request of process $i$. Under the well-formedness assumptions, process $i$ could not write to $l_j(pc)$. Similarly $t_j$ only accesses the memory area referenced by $t_i$ if $Complete(t_i, t_j)$, which has been assumed not to be the case.

To show commutativity, suppose $t_i$ is a Wait of process $i$, and $t_j$ is any transition of process $j$, $i \neq j$, and $t_i, t_j \in enabled(\sigma)$ for some $\sigma \in \Sigma$. Since $\neg Complete(t_i, t_j)$, $t_j$ does not post or complete another request that could cause communication between process $i$ and process $j$. As $t_i \in enabled(\sigma)$, there are two cases to consider: (i) When the communication has already happened, and (ii) when the communication has not yet happened but the corresponding request necessary to complete this communication has been posted (e.g., a matching Irecv for an Issend has been posted).

For case (i), the Wait operation updates only the local and global stores of process $i$. Under the assumptions any other operation would manipulate disjoint memory space. For case (ii), the Wait operation nondeterministically chooses between posted requests that Match the one being waited on that are not Completed in the current state. Upon choosing a request on process $k$ to form a communication the Wait marks the local and remote requests, transfers data from sender to receiver, and writes the local store of process $i$. If $t_j$ is any transition except for a Wait or Test on the request of process $k$ commutativity is trivial. Otherwise it has been
assumed that \( \neg \text{Complete}(t_i, t_j) \) so \( t_j \) could not be a Wait or Test on the request of process \( k \).

Lemma 5 If \( t_i \) is a Test of process \( i \) and \( t_j \) is any transition of another process \( j \) where \( i \neq j \) and \( \neg \text{Complete}(t_i, t_j) \) then \( I(t_i, t_j) \).

Proof. To show enabledness, it need only be the case that other processes are not allowed to write \( l_i(pc) \) (which is assumed to be the case). Now suppose \( t_j \notin \text{enabled}(t_i(\sigma)) \). Then \( t_i \) must have written \( l_j(pc) \) or \( t_j \) was a Wait on a request of process \( i \). Neither case is possible under the semantics and assumptions.

To show commutativity, the proof is essentially the same as that for Wait with a third case: (iii) When the communication has not yet happened and the corresponding request necessary to complete this communication has not been posted (e.g., the matching Irecv for an Issend has not been posted). Since \( \neg \text{Complete}(t_i, t_j) \), \( t_j \) does not post or complete another request that could cause communication between process \( i \) and process \( j \). Therefore the Test operation will write only the local store \( l_i \) of process \( i \). Thus making commutativity with all other transition types trivial.

The final theorem of this section combines all of the lemmas as a convenience for the proofs in the next section. It is interesting to note that the definition used here for independence implies the definition of independence used in [24].

Theorem 3 For all transitions of processes \( i \) and \( j \), \( t_i \) and \( t_j \), where \( i \neq j \) and states \( \sigma \in \Sigma \), \( t_i, t_j \in \text{enabled}(\sigma) \) and \( \neg \text{Complete}(t_i, t_j) \Rightarrow I(t_i, t_j) \).

Proof. This is concluded from Lemmas 1, 2, 3, 4, 5, the definition of the predicate \text{Complete} and the definition of the relation \( I \).
or rank of a process. (ii) In the presence of wildcard receive operations, it becomes
difficult to know whether all processes with possible Issends that could match a
given Irecv will be scheduled before the matching Irecv without fully expanding the
state. (iii) The transition dependencies are only between operations that \texttt{Complete};
therefore, one needs to know which Wait or Test operations are passed the request
handle returned by an Issend or Irecv.

The partial-order reduction algorithm proposed and implemented appears in
Figure 3.11. The DPOR algorithm adds PIDs of processes with enabled transitions
to the set \texttt{interleave} while exploring in the forward direction. Here, \(\varepsilon\) is the
choose operator \cite[Page 73]{46}. A state at \texttt{top}(s) on line 31 is popped from the
search stack \(s\) when that state has no enabled transitions or when there are no
more transitions to try according to the iterator from \texttt{nextinterleaved}. The
\texttt{nextinterleaved} procedure returns the next enabled transition for each process
in \texttt{interleave}(q), at state \(q\), allowing iteration through each enabled transition
of the process in \texttt{interleave}(q). Intuitively, these enabled transitions are held
in a list, where \texttt{nextinterleaved} causes the head of the list to be removed from
the list and returned at each invocation. Enabled transitions are added to the list
when a process is added to \texttt{interleave}. The operation \texttt{available}(q) is used to
denote the number of transitions held in \texttt{nextinterleave}(q). When a state \(q\) is
about to be popped from \(s\), the transition executed to generate \(q\), namely \texttt{tran}(q),
is compared to all transitions in the search stack using the predicate \texttt{Complete}.
The state \(v\) is the predecessor of the closest (see Figure 3.11, line 27) transition
\texttt{tran}(s[i]) such that \texttt{Complete(\texttt{tran}(q), \texttt{tran}(s[i])}) In state \(v\), process \texttt{proc}(q) that
executed to generate \(q\) is scheduled.

Stateful search and dynamic-partial order reduction are, unfortunately, not
orthogonal techniques. When a state is revisited, it is impossible to know whether
transitions that follow that state in the search have a dependent transition that
is currently on the search stack. On lines 19 – 23, the search stack is examined
for states generated by Issend, Irecv, Wait, or Test transitions, whenever a state
is revisited in the hash table. This is done using the \texttt{IsCommunication} operation
\[ q, q' : \text{state} \]
\[ s : \text{stack} \]
\[ h : \text{state set} \]

1. \( q := \text{initial state} \)
2. \( \text{interleave}(q) := \{ (\varepsilon \ p : \exists t_p \in \text{enabled}(q)) \} \)
3. \( \text{push}(s, q) \)
4. \( h := \{ q \} \)

5. \textbf{while} \( \text{size}(s) > 0 \)
6. \( q := \text{top}(s) \)
7. \( \text{if} \ \text{available}(q) > 0 \)
8. \( t_p := \text{next interleaved}(q) \)
9. \( q' := t_p(q) \) (* Local assertions are checked here. *)
10. \( \text{if} \ q' \notin h \)
11. \( h := h \cup \{ q' \} \)
12. \( \text{interleave}(q') := \{ (\varepsilon \ p : \exists t_p \in \text{enabled}(q')) \} \)
13. \( \text{push}(s, q') \)
14. \( \text{else} \) (* Cycles are detected here. *)
15. \( \text{if} \ \exists i \in \text{Dom}(s) : s[i] = q' \)
16. \( \text{report an error and exit} \)
17. \( \text{end if} \)
18. \( \text{for each} \ i \in \text{Dom}(s) \)
19. \( \text{if} \ i > 0 \land \text{IsCommunication}(s[i]) \)
20. \( \text{interleave}(s[i-1]) := \{ p : 0 \leq p < N \} \)
21. \( \text{end if} \)
22. \( \text{end for each} \)
23. \( \text{end if} \)
24. \( \text{else} \)
25. \( \text{if} \ \exists i \in \text{Dom}(s) : \text{Complete}(\text{tran}(s[i]), \text{tran}(q)) \)
26. \( v := \text{pre}(s[\max(i \in \text{Dom}(s) : \text{Complete}(\text{tran}(s[i]), \text{tran}(q)))]) \)
27. \( \text{interleave}(v) := \text{interleave}(v) \cup \{ \text{proc}(\text{tran}(q)) \} \)
28. \( \text{end if} \)
29. \( \text{pop}(s) \)
30. \( \text{end if} \)
31. \( \text{end while} \)

**Figure 3.11.** Dynamic partial-order reduction based stateful depth-first search.

that returns true exactly when \( \text{tran}(s[i]) \) is an Issend, Irecv, Wait or Test. If such a transition is found, the pre-state of this transition is fully expanded. This is a conservative step that is absent in stateless search.

A stateless search variant of the DPOR algorithm of Figure 3.11 is shown in Figure 3.12. A tradeoff is made to incorporate the stateful search. Here it is not necessary to make any assumptions with regards to the future of the search because every trace is maximally computed. However, fewer states may be visited by the
\( q, q' : \text{state} \)
\( s : \text{stack} \)
\( h : \text{state set} \)

1. \( q := \text{initial state} \)
2. \( \text{interleave}(q) := \{ \varepsilon \ p : \exists t_p \in \text{enabled}(q) \} \)
3. \( \text{push}(s, q) \)
4. \( h := \{ q \} \)
5. 
6. while size\((s)\) > 0
7. \( q := \text{top}(s) \)
8. if available\((q)\) > 0
9. \( t_p := \text{nextinterleaved}(q) \)
10. \( q' := t_p(q) \) (* Local assertions are checked here. *)
11. if \( q' \notin h \)
12. \( h := h \cup \{ q' \} \)
13. \( \text{interleave}(q') := \{ \varepsilon \ p : \exists t_p \in \text{enabled}(q') \} \)
14. \( \text{push}(s, q') \)
15. end if
16. else
17. if \( \exists i \in \text{Dom}(s) : \text{Complete}(\text{tran}(s[i]), \text{tran}(q)) \)
18. \( v := \text{pre}(s[\max(i \in \text{Dom}(s) : \text{Complete}(\text{tran}(s[i]), \text{tran}(q)))]) \)
19. \( \text{interleave}(v) := \text{interleave}(v) \cup \{ \text{proc}(\text{tran}(q)) \} \)
20. end if
21. \( \text{pop}(s) \)
22. end if
23. end while

**Figure 3.12.** A dynamic partial-order reduction based stateless depth first search model checking algorithm.

stateful algorithm of Figure 3.11 as revisited states are not re-explored.

The presentation will continue with a proof of correctness for the algorithm of Figure 3.11. The correctness of this algorithm depends upon the types of properties being preserved by the reduction. As stated in Section 3.2.1, the properties of interest are local assertions, the absence of deadlocks, and the absence of cycles.

**Theorem 4** If a local assertion is violated, a deadlock exists, or a cycle exists in the full reachable state space, then the violation, deadlock, or cycle will be detected by the DPOR algorithm of Figure 3.11.

**Proof.** (Sketch, the details are in Section 3.3.1.) The proof proceeds in two parts. The first part shows that the transitions explored by the DPOR algorithm form per-
sistent sets at each state — thereby preserving local assertions and deadlocks. Note that if no transitions are executed from a given state by the algorithm, then there must be no enabled transitions at that state. This empty set is therefore persistent trivially. If the set of enabled transitions is nonempty, then any transition that is enabled and dependent on the transitions selected from a given state $q$ but not executed at $q$ will eventually be executed by the DPOR algorithm. When backing off from the dependent transition, the algorithm will discover the dependency and schedule the process with the dependent transition at state $q$.

The second part of the proof shows that a cycle may be delayed but will not be ignored. Since all cycles are considered errors, when the cycle is eventually closed, it is reported at line 17.

3.3.1 Proof of Correctness of the Reduction Algorithm

This section presents a detailed proof of Theorem 4. Some definitions from [30], including some notations are presented first. Let $a \xrightarrow{t} b$ denote a transition $t$ of some process from state $a$ to state $b$. Let $a = a_1 \xrightarrow{t_1} a_2 \xrightarrow{t_2} \cdots a_n \xrightarrow{t_n} a_{n+1}$ denote a sequence of $n$ transitions of processes from state $a$ to $a_{n+1}$. Let $A_G$ denote the set of all reachable states of the model. Recall the definition of independence from Page 59.

**Definition 6** [30] The set of transitions $T$ enabled in state $q$ is persistent in state $q$ if and only if for all nonempty sequences of transitions

$$q = q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} \cdots q_{n-1} \xrightarrow{t_{n-1}} q_n \xrightarrow{t_n} q_{n+1}$$

from $q$ in $A_G$ and including only transitions $t_i \notin T, 1 \leq i \leq n$, $t_n$ is independent with all transitions in $T$.

**Definition 7** [30] A set $T$ of transitions is a conditional stubborn set in state $q$ if $T$ contains at least one enabled transition, and if for all transitions $t \in T$, the following condition holds: for all sequences $q = q_1 \xrightarrow{t_1} q_2 \xrightarrow{t_2} q_3 \cdots q_{n-1} \xrightarrow{t_{n-1}} q_n \xrightarrow{t_n} q_{n+1}$, of transitions such that $t$ and $t_n$ are dependent, at least one of the $t_1, \ldots, t_n$ is also in $T$. 
The proof of correctness for the DPOR algorithm will show that for all states visited by the DPOR algorithm, if the set of enabled transitions is non-empty in any state visited by the DPOR algorithm, then the set of transitions explored by the DPOR algorithm at that state is a conditional stubborn set [88]. As any cycle will be considered an error, conditional stubborn sets are sufficient to preserve the properties of interest.

**Lemma 6** When a state is popped from the search stack at \texttt{pop(s)}, if that state is reached through an Issend, Irecv, Wait, or Test operation of process \texttt{p}, \texttt{p} is added to the set \texttt{interleave} of the predecessor of the nearest dependent transition in the search stack.

**Proof.** When \texttt{Complete(t_i, t_j)} evaluates to false, the two transitions \texttt{t_i} and \texttt{t_j} are independent. However, when \texttt{Complete(t_i, t_j)} evaluates to true it is unknown whether \texttt{t_i} and \texttt{t_j} are Independent and so pessimistically, they are assumed to be dependent. Lines 26 – 30 of Figure 3.11 clearly show the DPOR algorithm performing a search through \texttt{s} for the nearest transition such that \texttt{Complete} evaluates to true. When such a state \texttt{v} is found, \texttt{p} is added to \texttt{interleave(v)}, thus making all enabled transitions of \texttt{p} at \texttt{v} available through \texttt{nextinterleaved(v)}.

By executing \texttt{p} at \texttt{v}, the DPOR algorithm will again search through \texttt{s} to find the nearest transition such that \texttt{Complete} evaluates to true. This will continue until no such transition is found. By using the predicate \texttt{Complete}, the dependence relation is over-approximated. This guarantees that \texttt{p} will not be scheduled in states in \texttt{s} only when it can be shown conclusively that the two transitions \texttt{t_i} and \texttt{t_j} are in fact Independent.

Before stating the correctness theorem for the DPOR algorithm and its accompanying proof, some additional notation is introduced. Although \texttt{s} is the search stack, it can also be viewed as a sequence of transitions from the initial state, where the top of the stack \texttt{top(s)} is the last state in the sequence. Let \texttt{s.w} denote the concatenation of transition sequence \texttt{s} with transition sequence \texttt{w}. Let the set member operator \(\in\) be overloaded for sequences. Further, a Wait or Test may
correspond to an Issend or Irecv. This means the index of the request created by the Issend or Irecv transition is passed to the Wait or Test — hence they (e.g., the Issend and Wait) correspond as they refer to the same request in the global store of the process executing both transitions.

Figure 3.13 is representative of a situation that may arise where there is a dependency between processes. The transitions $t_i$ and $t_c$ are both Wait transitions corresponding to Issend operations that Match transition $t_r$. The DPOR algorithm must detect this dependency and schedule $\text{proc}(t_i)$ and $\text{proc}(t_c)$ such that both $t_i$ and $t_c$ are eventually executed along some path explored by the DPOR algorithm.

**Theorem 5** At line 31 when a state is popped from the search stack $s$, the set of transitions $T$ explored by the algorithm of Figure 3.11 is persistent in $\text{top}(s)$.

**Proof.** Let $\text{top}(s)$ be denoted $q$. Also, let the set of transitions executed from $q$ be denoted $T$.

There are two cases: $T = \emptyset$, and $T \neq \emptyset$. If $T = \emptyset$ then $\text{enabled}(q) = \emptyset$ on lines 2 or 13. When $T = \text{enabled}(q)$ the set is trivially persistent in $q$.

It is shown in [30] that a conditional stubborn set [88] in some state $q$ is persistent in $q$. It will be shown that when $T \neq \emptyset$, $T$ is a conditional stubborn set and therefore persistent in $q$. The proof will proceed by contradiction. Suppose there is some state $q$ that does not form a conditional stubborn set of transitions that is explored by the algorithm. Let $w$ be a minimum length sequence of transitions of length $n$ such that (i) $\forall i \in 1 \cdots (n + 1): q_i \in s.w \Rightarrow q_i \in A_G$, (ii) for each $i$, the transitions

$$
\text{proc}(t_n) \text{proc}(t_r) \text{proc}(t_c)
$$

![Figure 3.13](image)

**Figure 3.13.** Three processes executing concurrently.
$t_i \in w$ are not in $T$ (where $T$ is the set of transitions executed at $q$), and (iii) $\text{last}(w) = t_n$ is dependent with some transition $t \in T$, i.e., $\neg I(t_n, t)$.

By the rule of contrapositive (i.e., the contrapositive of $\neg \text{Complete}(t_n, t) \Rightarrow I(t_n, t)$), $\neg I(t_n, t) \Rightarrow \text{Complete}(t_n, t)$; since $t_n$ and $t$ are assumed dependent, $\text{Complete}(t_n, t)$ is concluded. Therefore, both $t_n$ and $t$ are each either an Issend, Irecv, Wait, or Test. Now there are three of cases: (Case 1) $\text{proc}(t_n)$ is enabled at $q$ and remains enabled along some path executed by the DPOR algorithm, (Case 2) $\text{proc}(t_n)$ is enabled at $q$, but becomes disabled along all paths executed by the DPOR algorithm, (Case 3) $\text{proc}(t_n)$ is disabled at $q$ and remains disabled along all paths executed by the DPOR algorithm.

Case 1: Suppose $\text{proc}(t_n)$ is enabled at $q$ and remains enabled along some path executed by the DPOR algorithm. If $\text{proc}(t_n)$ is enabled at $q$ and remains enabled, then either $t_n$ will eventually be executed by the DPOR algorithm at some subsequent successor of $q$ (as it has been assumed the transition relations of the processes are finite), or a state will be revisited in the hash table. If $t_n$ is eventually executed, by Lemma 6, $\text{proc}(t_n) \in \text{interleave}(q)$. Therefore there exists some $i$ such that $t_i \in w$, $\text{proc}(t_i) = \text{proc}(t_n)$, and $t_i \in T$. Therefore $T$ is a conditional stubborn set, and a contradiction has been reached as it has been assumed that none of the transitions $t_i \in w$ were in $T$. Otherwise if a state is revisited in the hash table, all states on the stack that have out-going communication transitions will be fully expanded, including $q$ (recall that $t$ must be either an Issend, Irecv, Wait, or Test transition). Therefore at least some $t_i \in w$ is included in $T$ (where $\text{proc}(t_i) = \text{proc}(t_n)$), making $T$ a conditional stubborn set, resulting in contradiction as it has been assumed that none of the transitions $t_i \in w$ were in $T$.

Case 2: Suppose $\text{proc}(t_n)$ is enabled at $q$ but becomes disabled in all paths explored by the DPOR algorithm. Let $t_i \in w$ such that $\text{proc}(t_i) = \text{proc}(t_n)$ where $i < n$ be the transition of $\text{proc}(t_n)$ that becomes
disabled along every path explored by the DPOR algorithm. It has been assumed that each Issend and Irecv must be followed by a Wait or Test. By inspection of the formal semantics, only a Wait corresponding to an Issend (\texttt{wait h where h = issend a b} was executed previously by the same process) such that the Issend matches a wildcard Irecv can become disabled after having been enabled. Furthermore, such disabling can happen only by the execution of the Wait or Test corresponding to an Issend of another process that matches the same wildcard Irecv request. From these facts, \( t_i \) is a Wait on an Issend and a conflicting Issend exists. Let the corresponding Wait or Test on the conflicting Issend transition be \( t_c \) (\( c \) indicating conflict). This is depicted in Figure 3.13. Therefore, along every path explored by the DPOR algorithm, there is an occurrence of \( t_c \). Since \( t_i \) and \( t_c \) are both enabled in \( \text{pre}(t_c) \) and at least \( t_i \) is a Wait on an Issend \( (t_c \) could be a Wait or a Test on an Issend), there must be some wildcard Irecv transition \( t_r \) that happens before \( t_c \) along every path that is explored by the DPOR algorithm.

Since \( t_r \) is an Irecv that matches both \( t_i \) and \( t_c \), \( \neg I(t_r,t_c) \), and therefore \texttt{Complete}(\( t_r,t_c \)). It follows that \texttt{proc}(\( t_c \)) \( \in \text{interleave}(\text{pre}(t_r)) \), and therefore the DPOR algorithm explores at least one execution trace where \( t_c \) happens before \( t_r \). There are two cases: (Case 2.a) where \( t_r \) happens before \( q \) and (Case 2.b) where \( t_r \) happens after \( q \); however, it must be the case that the corresponding Wait or Test for \( t_r \) happens after \( t_c \) and hence after \( q \). Therefore, the proofs for Cases 2.a and 2.b are substantively the same. The Case 2.b where \( t_r \) happens after \( q \) will be presented.

Now consider two cases: Case 2.b.i: \( t_c \) could be a Wait, or Case 2.b.ii: \( t_c \) could be a Test.

\textbf{Case 2.b.i:} Suppose \( t_c \) is a Wait. If \( t_c \) is a Wait, it will be disabled in a
state before \( t_r \), however, \( \text{proc}(t_c) \) can still be enabled up to \( t_c \). It has been assumed that a Wait or Test corresponding to \( t_r \) will eventually follow \( t_r \) along all paths in \( \mathcal{A}_G \). Let this operation be \( t_w \). Now the DPOR algorithm will explore the execution of \( t_w \) and \( t_c \) in both orders. When the DPOR algorithm chooses the order where \( t_w \) happens before \( t_c \), \( t_w \) will create at least two successor states, one of which will choose the request created by the Issend corresponding to \( t_i \), guaranteeing that \( \text{proc}(t_n) \) cannot be disabled from this state. Therefore the DPOR algorithm eventually explores \( t_i \), a contradiction as it has been assumed that \( \text{proc}(t_n) \) becomes disabled before \( t_i \) is executed and remains disabled along every path explored by the DPOR algorithm.

**Case 2.b.ii:** Suppose \( t_c \) is a Test. If \( t_c \) is a Test, it will be enabled in a state before \( t_r \) is executed. Since \( \text{Complete}(t_c,t_r) \), the DPOR algorithm will explore \( t_c \) and \( t_r \) in both orders. When the DPOR algorithm explores the trace where \( t_c \) happens before \( t_r \), executing \( t_c \) will cause the flag referenced by \( t_c \) to be set to \( false \), and update the PC of \( \text{proc}(t_c) \). The execution of \( t_c \) in this state will not be able to complete a communication as the request corresponding to \( t_r \) will not have been posted. When the Wait or Test \( (t_w) \) corresponding to \( t_r \) is executed by the DPOR algorithm, it will result in at least two successor states, one of which will choose the request created by the Issend corresponding to \( t_i \), guaranteeing that \( \text{proc}(t_n) \) cannot be disabled from this state. Therefore the DPOR algorithm eventually explores \( t_i \), a contradiction as it has been assumed that \( \text{proc}(t_n) \) becomes disabled before \( t_i \) is executed and remains disabled along every path explored by the DPOR algorithm.
Therefore, it is not possible for \texttt{proc}(t_n) to remain disabled along all paths explored by the DPOR algorithm.

\textbf{Case 3:} Suppose \texttt{proc}(t_n) is disabled at \texttt{q} and remains disabled along all paths explored by the DPOR algorithm. Then, there is a sequence of \(m\) transitions in \(w\) \((m > 0)\), where for each \(j\) in \(1 \ldots m\), \(t_j \in w\) must execute to enable \(t_n\) in \(q_n\). Let this be the \textit{enabling sequence} of transitions. The posting of requests, i.e., executing Issend or Irecv transitions are the only transitions that could enable a disabled transition (recall only Wait can be disabled). The first \(t_j\) in the enabling sequence must therefore be either an Issend, or an Irecv. Since each of these transition types are always enabled, \texttt{proc}(t_j) is enabled at \texttt{q} and will eventually be executed by the DPOR algorithm. Therefore the enabling sequence \texttt{proc}(t_n) along some path from \texttt{q} that is explored by the DPOR algorithm, a contradiction with the assumption that \texttt{proc}(t_n) remains disabled.

Therefore if there is a sequence of transitions \(w\) from \texttt{q} where some transition \(t \in T\) and \texttt{last}(w) are dependent, then some transition \(t_i \in w\) is also in \(T\).

The MPI standard requires that all processes eventually call Finalize. A cycle in the full state space will prohibit processes from reaching Finalize and therefore is an error. the DPOR algorithm shown in Figure 3.11 checks for cycles while performing the model checking. It is therefore desirable to show that the DPOR algorithm actually detects the presence of cycles.

\textbf{Theorem 6} \textit{The DPOR algorithm of Figure 3.11 discovers a cycle in the reduced state space if and only if there is a cycle in \(A_G\).

\textbf{Proof.} In the \(\Rightarrow\) direction, the proof is trivial since all transitions explored by the DPOR algorithm are in \(A_G\). For the \(\Leftarrow\) direction, suppose there exists some cycle in \(A_G\) that is not explored by the DPOR algorithm. Let \(w\) representing the least sequence in \(A_G\) that contains the unexplored cycle be as follows:
\[ w = q_0 \xrightarrow{t_0} q_1 \xrightarrow{t_1} q_2 \cdots \xrightarrow{t_n} q_n \]

where there exists some \(0 \leq i < n\) such that \(q_i = q_n\) for some \(q_i \in w\). Now let \(j\) be the maximal transition index such that the DPOR algorithm explores the prefix of \(w\), \(t_0 \ldots t_j\) (the DPOR algorithm explores at least the initial state \(q_0\)). Theorem 5 shows that in \(q_j\) the set of transitions that are explored by the DPOR algorithm are a persistent set in \(q_j\). The next transition \(t_{j+1}\) may not be explored by the DPOR algorithm at \(q_j\) because \(t_{j+1}\) is enabled but independent of all other transitions explored from \(q_j\). Therefore, \(t_{j+1}\) remains enabled in the next state and conclude that the DPOR algorithm eventually explores \(t_{j+1}\). Let \(q'\) be the state generated by the DPOR algorithm upon executing \(t_{j+1}\). Now there are three cases: either (i) \(q'\) closes the cycle, detected by lines 16 – 18 of the DPOR algorithm, (ii) there are more transitions in \(w\), or (iii) the DPOR algorithm revisits a state in the hash table that is not currently on the search stack. For Case (i), if the cycle is closed the proof is complete. For Case (ii), if there are more transitions in \(w\), the process can iterate until the last transition in \(w\) is visited. At this point every transition in \(w\) has been explored by the algorithm and the final state is a revisit in the search stack as all of the state transitions necessary to create the cycle have been taken by the DPOR algorithm. For Case (iii), the DPOR algorithm revisits a state \(v\) in the hash table that is not currently on the search stack. Let this be the \(n^{th}\) revisit of \(v\). On the first visit to \(v\), a persistent set of transitions was computed and explored by the DPOR algorithm before \(v\) was removed from the search stack. Therefore, either case (i) or case (ii) would have applied and the cycle would have been discovered by the DPOR algorithm earlier.

\[ \square \]

3.4 Experiments Using DPOR for MPI

To demonstrate the effectiveness of the specialization of partial-order reduction for MPI primitives, consider the two-dimensional diffusion simulation described in [78]. Here the authors model the MPI primitives using Promela and attempt to model check for up to a \(4 \times 4\) grid (16 processes). They report that the SPIN model checker runs out of memory but is able to handle a \(4 \times 3\) array. It is not clear whether
the authors attempt to use the partial-order reduction implemented in SPIN.

To handle this program several modifications were made. First the pseudo-code shown in their paper was implemented in C. Then, the program was transformed so that all of the Send andRecv operations were the corresponding Issend and Irecv operations followed by a Wait. Optimization came next — the MPI operations were moved such that setting up buffers for communication could overlap the sending and receiving of buffers.\(^2\)

From the modified program text a model was automatically extracted using the framework which extends the Microsoft Phoenix compiler [57]. The program model is then automatically simplified by inlining and slicing such that only the communication skeleton is preserved as described in Chapter 4. Only a \(2 \times 2\) configuration could be verified by the MPIC tool, generating 19M states in 10 minutes on a laptop computer (2GHz., 2GB Ram). This example could not be verified at all without the dynamic partial-order reduction model checking algorithm. The next section discusses one more example which elucidates the inner workings of the DPOR algorithm even more.

### 3.5 A Detailed Example

The algorithm is similar in many respects to the DPOR algorithm proposed in [24], with one notable difference. In the Godefroid and Flanagan work, when a process \(p\) is added to the backtrack set \(\text{backtrack}(v)\) for some state \(v\), the DPOR algorithm checks to insure that \(p\) is enabled. If \(p\) is disabled, a check is performed to find some transition \(t \in \text{enabled}(v)\) such that \(p\) becomes enabled in \(t(v)\). If no such \(t\) can be found \(v\) is fully expanded. The algorithm does not require this check as a result of the semantics of MPI. In particular, only Wait can become disabled after it is enabled.

To better understand the behavior of Wait, consider the execution sequence of the model checker using stateless search when the example in Figure 3.14, is instan-

\(^2\)This program code can be downloaded from the Internet at http://www.cs.utah.edu/formal_verification/verification_environment.
if(rank == 0){
    h = Irecv * (addrof x);
} else {
    h = Issend 0 (addrof x);
}
Wait h;

Figure 3.14. A program pseudo-code fragment containing a non-deterministic receive operation.

tiated for three processes. If the communication execution order is 0 : h = Irecv *;
1 : h = Issend 0; 1 : Wait h; 2 : Issend 0; 0 : Wait h; then the problem is that
the Irecv operation may not form a communication with the Issend of process 2
because 1 : Wait h will force a match and disable the ability of 0 : Wait * to form a
communication nondeterministically under the execution semantics. However after
executing the DPOR algorithm, the interleave sets would be produced as shown
in the state nodes of Figure 3.15.

The nodes represent states as they would be visited by the stateless search
variant of Figure 3.12. The states are numbered in order that they are first
visited. The exploration up to state 14 is shown. The 2 : h = issend 0 is eventually
executed above the 0 : wait h in state 3, resulting in state 9. It remains to show
that some interleaving beyond state 9 will either have 2 : wait h or 0 : wait h
happen before 1 : wait h. This is the case because 0 : wait h and 1 : wait h are
guaranteed to happen in both orders whenever they appear along any path because
Complete(0 : wait h, 1 : wait h).

3.6 Related Work

This chapter has focused on the presentation of a partial-order reduction algo-

rithm that is customized for the communication semantics of MPI. Other models
of MPI exist including [77] where the authors build a model of MPI from first
principles. They then propose an urgent scheduling for MPI operations included
in their model that preserve a set of halting properties. They have implemented
many of the MPI primitives — including a number of nonblocking operations in
the SPIN [76] model checker. The approaches are difficult to compare because they work for different subsets of MPI. It is also not clear how or whether partial-order reduction is being used in their current implementation.

Other previous work in formalizing MPI such as [6, 27, 65, 66, 77] do not implement the semantics proposed directly in a model checker. Rather these models serve to augment the program model in a library format.
There are several model checkers that have partial-order reduction such as SPIN [36], Zing [2], Verisoft [31], Bogor [73] and perhaps others. In each of these, the reduction is not tailored to MPI.

Partial-order reduction has been studied extensively — a survey of which is beyond the scope of this chapter. This work is most closely related to the dynamic partial-order reduction algorithm of Godefroid and Flanagan [24]. The three primary differences being (i) the algorithm is tailored for MPI operations, (ii) there is no place in the algorithm where it is necessary to search for an enabling process, and (iii) the algorithm uses stateful search.

Model checking (cf. [13, 15]) is an automata-theoretic approach to automatic program verification. Given some model of a system, all of the possible configurations or states of the system are enumerated. Logical properties can then be checked on states (safety properties [43]), paths (linear-time temporal logic or LTL [90], or trees (branching-time temporal logic or CTL [8]) in the state graph.

This work is primarily concerned with explicit state model checking technology such as that implemented in SPIN [36], PV [60], Murphi [16], Zing [3], and others. Explicit state model checkers represent each of the visited system configurations as some structure of state elements. This is contrasted to so-called symbolic model checkers [54] where the set of visited states are represented using BDDs [11] or other boolean representations.

Partial-order reduction is a technique used to find and check representative sequences [71] while model checking, instead of generating the full state space. These sequences, sometimes called Mazurkiewicz traces [52], represent the set of possible traces created by permutations of the transitions in the sequence under some commutativity relation. If the property is found to hold over all representative traces, then the property holds over the full state graph. Similarly when there is a property violation in the state graph, then at least one of the representative sequences will also violate the property being checked.

Significant work has been done in the area of partial-order reduction. Valmari [85, 86, 87, 88, 89] proposed the theory of stubborn sets, which were later
shown to be equivalent to persistent sets of Godefroid [28, 29, 30]. A persistent set $T$ at some state $\sigma$ (described in detail in Chapter 3) is a set of transitions such that the transitions in $T$ are independent with respect to those transitions not in $T$. Persistent sets form the basis for conditions $C0$ and $C1$ of [15].

Partial-order reduction can be combined with temporal logic model checking and computed on-the-fly [69, 70, 39], or in the process of building the state graph, thus making it unnecessary to build the full state graph. The guards of a model can be strengthened [45, 44], thus effecting a reduction.

More recent work in partial-order reduction include [7] where the authors generalize the idea of visibility thereby allowing clusters of processes to commute. In [10] a breadth first search is adapted to use partial-order reduction. The authors of [9] extend partial-order reduction for concurrent systems without well defined process boundaries. In addition to the above work, [1, 41, 48] combines the theory of partial-order reduction with symbolic model checking.

Other views of reduction have been based on the commutativity of right and left moving transitions first proposed by Lipton [50]. Nalumasu et. al. [61], propose an $LTL_\neg X$ preserving reduction based on this view. In [23] the authors use movers to determine which regions of code are atomic. In [49] the authors determine regions that could be atomic under the commutativity of movers, called transactions, and model check with respect to these transactions.

Independence of transitions, used to define commutativity in partial-order reduction frameworks, is defined in terms of transitions and states. When the semantics of each transition is obvious (say, any transition reads or writes a single memory location), it is easy to determine when two transitions are independent. Unfortunately, the semantics of MPI are far from obvious; in fact, apart from an early LOTOS specification for a small subset of MPI [27], or the specification of some MPI operations including send, receive, etc. [77] using automata communicating through channels, there exists neither a more direct (e.g., transition system based) nor extensive formal characterization for MPI. MPI communication commands include dozens of flavors of sends and receives. The send command
always has to specify the receiver, but the *receive* command may or may not specify
the sender. The former case is that of a *nonwildcard* receive, and the latter case a
*wildcard* receive. Wildcard receives can match any sender targeting the process in
which the receive occurs (barring *tag-matching*, a detail suppressed in this chapter).
In addition, one finds various forms of *wait* and *test* commands that help await
(blocking semantics) or test (probe and return true/false without blocking) for the
completion of asynchronous message transmittal. There are also many collective
operations such as *barriers* and *reduces*.

Past work in the formal analysis of MPI includes work by Siegel and Avrunin [75,
76, 77, 78, 79] and at The University of Utah [6, 65, 66, 72]. To make the
contrast between past work and what is proposed here clearer, consider the two
main commands supported by MPI, namely *send* and *receive*. In [77, 79], Siegel
and Avrunin study “ordinary” flavors of these commands (known as *MPI_Send*
and *MPI_Recv*). The MPI run-time system is allowed (but not required) to provide
buffering for an *MPI_Send*. With buffering, the sender can post the message and
asynchronously proceed; without buffering, a rendezvous-based message exchange
happens. In [77, 79], Siegel and Avrunin show that one can analyze wildcard-free
MPI programs for the absence of deadlocks by merely considering rendezvous-based
executions, thus not inviting state explosion by modeling various degrees of the
asynchronous sending. In [75], the authors propose the following extension to
their earlier result: in an MPI program with wildcards, if all senders that can
potentially match a wildcard receive are known, then the same rendezvous-style
communication can be forced. The knowledge about the senders comes in two
ways: (i) all these sends are found as moves out of the current global state *s*
being examined during depth-first search based explicit state enumeration model
checking, or (ii) only some of the sends are found as moves out of *s*; in this case
it is assumed that the remaining sends will *never* be found offered in all future
computations from *s* (thus, what is offered can be considered the *ample set* [15]).

---

3Note that a very similar result was obtained in another setting by Manohar [51] and captured
as the *Slack Elasticity* theorem.
The authors propose a so-called urgent algorithm based on these ideas.

While shown effective on many case studies, the urgent algorithm only provides a partial answer to how partial-order reduction algorithms may be designed for MPI. First of all, the urgent algorithm was defined with respect to a formal model of MPI communication (introduced in [77]) that models only a few MPI functions. Many functions, especially the deterministic flavors of MPI send and receive, namely MPI_Issend and MPI_Irecv (frequently used during MPI program optimization) are not modeled. These commands behave in a deterministic manner because the associated communication buffers are explicitly provided by the user. They are termed non-blocking MPI calls, because after posting the send/receive, the computation advances without blocking, relying on a subsequent wait/test to check for message transmittal. Although a recent paper [76] considers non-blocking MPI commands, no partial-order reduction method is proposed for these non-blocking operations. In short, the approach taken in the urgent algorithm cannot be used as the basis of partial-order reduction for most MPI functions because: (i) a general understanding of the MPI semantics is required, and (ii) dependency on the future must be handled in a general manner. In this chapter, a partial-order reduction algorithm for MPI based on more general design principles is proposed. More specifically, the communication semantics of MPI are formalized, and then state and prove the classical notion of independence [15] among MPI program commands with respect to the formal semantics.

The behavior of wildcard receives (that future sends may induce a dependency on wildcard receives) is emblematic of a much more general issue with respect to MPI (and concurrent program analysis in general). It is well known that dependencies between operations will be precisely known only at runtime. This is one of the main reasons why Flanagan and Godefroid devised the dynamic partial-order reduction (DPOR) method [24] for general concurrent software analysis. While the specific examples that motivated Flanagan and Godefroid pertained to unknown aliasing relationships and array range overlaps, the same thinking can be applied to the statically unknown (but dynamically known) dependency information in a
communication oriented language such as MPI. The possibility of handling many more dynamic features of MPI appears within reach — such as MPI_Cancel — which allows a pending MPI operation to be canceled.

In the context of DPOR itself, the algorithm has three primary differences: (i) the DPOR algorithm is tailored for MPI operations in some natural ways, (ii) when using stateless search, there is no place in the DPOR algorithm where there is no need to perform full expansion of a state to deflect unsoundness, and (iii) the algorithm combines stateful search. Regarding point (i), the MPI standard requires that correct MPI programs terminate at an MPI_Finalize call. Hence, the acyclic state-space requirement of MPI is checked by default in the algorithm, which keeps fingerprints of visited states in a hash table. For point (ii) this discussed in detail with respect to the example shown in Figure 3.14.

3.7 Summary

This chapter has presented a modeling language for MPI, and a simplified view of MPI, having only Issend, Irecv, Wait, Test, and Barrier operations with the presence of wildcards receives. Under this simplification a number of independence theorems have been proven regarding the transition types of the MPIC modeling language.

A dynamic partial-order reduction algorithm is presented that uses the aforementioned independence theorems to effect a greater reduction than would be possible otherwise. Correctness theorems with regards to proposed DPOR algorithm are proved to show that the algorithm preserves deadlocks, local assertions, and the presence of cycles.
CHAPTER 4

MODEL EXTRACTION AND SIMPLIFICATION

Simplification of program models is common in the application of model checking analysis. There is a need for simplification as the actual program may be too large, or use language features that are too complicated to model check within the allotted execution time or memory space.

Presently the authors are not aware of simplification techniques customized for SPMD MPI programs. This chapter proposes a modeling framework and a set of program simplifications that remove the majority of program code while preserving local assertions, deadlocks, and the presence of cycles. The simplifications are: (i) inline all user defined functions, (ii) remove unknown program constructs, (iii) slice the model with respect to MPI operations and user placed assertions, and (iv) remove empty counting loops from the sliced model.

It is not possible to guarantee that all assertions that a user might write are preserved by these simplifications. In particular, removing unknown program constructs — such as calls into the standard C library, and the removing of loops that the heuristic recognizes as empty counting loops, may mask assertion violations, cycles, and deadlocks. However this work does assert that the inlining of functions and the slicing algorithm do not remove behaviors from the model that are visible to the properties being checked. The burden is therefore placed on the user to understand these simplifications and apply them judiciously.

The chapter is organized as follows: Section 4.1 presents an overview of the modeling framework. The remainder of the chapter focuses on the details of extracting a model and applying the simplifications necessary to facilitate model checking. The MPIC intermediate representation (MPIC IR) is described in Section
4.2. The program graph traversal algorithms are described in Section 4.3 along with an overview of the transformation operations implemented that use the traversal algorithms. How different models are produced is contained in Section 4.4. Related work is covered in Section 4.5, along with some discussions of error trail replay. Section 4.6 summarizes the chapter.

4.1 An Overview of the Modeling Framework

The modeling framework uses the Microsoft Phoenix [57] compiler as a front end. The Phoenix compiler allows developers to insert a compilation phase between existing compiler phases in the process of lowering a program from language independent MSIL (Microsoft Intermediate Language) to device specific assembly. A phase is inserted at the point where the input program has (i) been simplified into a single static assignment (SSA) form, with (ii) a homogenized pointer referencing style that is (iii) still device independent. This phase reads the Phoenix intermediate representation and builds from it a state-transition system (the MPIC IR) for each function, which is similar in spirit to a control flow graph. Control locations in the program are represented by states, and program statements are represented using transitions.

The architecture of the verification framework is shown in Figure 4.1. The user may input a program in any language that can be compiled using the Phoenix back end — this work has experimented only with C. The program is compiled into an intermediate representation, the Phoenix IR. The Phoenix IR is read to create a separate intermediate program representation, the MPIC IR. From the MPIC IR, different formats can be output, including TLA+, DOT [21], and MPIC (described in Chapter 5). The framework integrates both TLC and a new model checker MPIC to perform the verification tasks. If an error is found, the error trail is then made available to the verification environment, and can be used by the framework to drive the Visual Studio debugger to replay the trace to the error. The remainder of this section describes the simplification and replay capabilities of the framework.
4.1.1 Program Simplifications

From the extracted state-transition format, it would be possible to emit a TLA+ model directly. However, the TLA+ model would require the addition of sufficient mechanisms to handle function calls and returns. Although this is possible with TLA+ (such as is done in [47]) — many scientific computing applications would not benefit from the additional functionality. As such, this work proposes the following sequence of transformations, intended to reduce the complexity of model checking while preserving the properties of interest, before applying model checking based analysis. The simplifications are as follows:

- Inline all user defined functions: It is assumed (i) that all parameters are pass by value, (ii) there are no function pointers, and (iii) there is no recursion.

- Remove operations foreign to the model checking framework: Examples include `printf`.
• Slice the model with respect to communications and user assertions: The cone of influence of variables is computed using a chaotic iteration over the program graph, similar to what is described in [63].

• Eliminate redundant counting loops: This is a heuristic to handle computational loops that occur frequently in MPI programs.

4.1.2 Program Modeling

The models of MPI from the previous chapters are intended to capture the communication semantics while abstracting away the possible implementation details. However, there are some implementation details retained that are common to all present-day computer systems, and that are implied by the MPI 1.1 standard. When modeling C programs in TLA+ or MPIC, these details must be considered. It is assumed that each process operates in a disjoint memory space. As such, in a TLA+ program model, an array of variables is allocated to represent the local store of each process. More formally, memory is modeled as a function $\text{Memory} : \mathbb{N} \rightarrow \mathbb{N}$ where allocated addresses are mapped onto values. Variable names are represented by an array of addresses that use symbols (i.e., strings) for indices. These are again functions that map strings onto addresses. The mention of memory brings to the fore the first of several simplifications that are imposed on the model. Only values in $\mathbb{N}$ are considered valid memory contents. With an explicit notion of memory and addresses, it is possible to have explicit pointers in the model. This is supported, allowing for arbitrary dereferences. Address 0 is never allocated, allowing for null pointer dereference violations to be discovered.

It is possible to allocate memory using the operator in Figure 4.2. This operator updates the function representing process memory by changing the $\text{Memory}$ function for process $i$ such that there are $\text{size}$ new memory locations at the end, each having uninitialized memory contents. The operator also writes the address of the first uninitialized location into the memory location of the pointer.

Many constants are used by MPI and consequently in the model. Since the program model is automatically extracted from the program while it is being
Figure 4.2. A TLA+ rule for memory allocation used with MPI program models.

compiled, it is necessary that the constants used in the model match those used by
the implementation of MPI used with the program being analyzed. These constants
are provided in a separate TLC configuration file. The constant definitions generally
match the values used in the corresponding C header files (mpi.h). Since not all
values can be used (e.g., no floating point values, etc.) some manual changes to the
configuration and corresponding header files can be made if necessary.

The individual transitions are formatted as shown in Figure 4.3, combined with
the initial values of the memory array and the map from variable names to their
addresses and written to disk. The constants, program model, and MPI model
are then given to the TLC model checker. The TLA+ model corresponding to the
program of Figure 1.2 is contained in Appendix B.

\[
\begin{align*}
& pc[pid] = \text{state}_pc \\
& pc' = [pc \text{ EXCEPT } ![\text{pid}] = \text{next}_pc] \\
& \text{guard} \\
& \text{action} \\
& \text{UNCHANGED <<program variables not written in the action>>}
\end{align*}
\]

Figure 4.3. A Transition template for TLA+ program models.
4.1.3 Error Trail Generation

In the event that the model contains an error, an error trail is produced by the model checker and returned to the verification environment. To map the error trail back onto the actual program, changes in the error trail to variable values that appear in the program text are observed. For each such change, the Visual Studio debugger is stepped until the corresponding value of the variable in the debugger matches. The process that moves at every step is also observed in the error trail and context switch between processes in the debugger at corresponding points are imposed. When the error trail ends the debugger is within a few steps of the error, with the process that causes the error scheduled.

4.1.4 Examples

The semantic evaluation framework has been applied to a small number of examples and show the results of a few verification tasks in this section. Table 4.1 shows the number of states generated / execution time for the following examples: (i) an example code from [64], (ii) the 2D diffusion example from [78], and (iii) the last scenario described in Section 1.2 and coded in Figure 1.2, namely “What is guaranteed about the matching receive when the first send completes?” Each of the experiments was run on a dual core 2GHz processor with 2GB of memory. When TLC was applied, two worker threads were used.

The Trap example from [64] computes the integral over a trapezoidal region. The program is written in the SPMD style and is typical of “textbook examples” in this area. The example, as written, is verified for the absence of deadlocks and the default assertions provided by the respective model checkers for two model processes.

Table 4.1. The number of states generated and execution time in seconds on some simple examples.

<table>
<thead>
<tr>
<th></th>
<th>MPI-TLC</th>
<th>MPIC without DPOR</th>
<th>MPIC with DPOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trap</td>
<td>724/2</td>
<td>331/0</td>
<td>66/0</td>
</tr>
<tr>
<td>Diffusion 2D</td>
<td>timeout</td>
<td>timeout</td>
<td>19,807,253/623</td>
</tr>
<tr>
<td>Scenario 4</td>
<td>310/2</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The Diffusion 2D example computes the diffusion of a substance through a two-dimensional grid of processes. The grid is a $2 \times 2$ array of processes where each process houses a subset of the cells. The framework could not verify the pseudo-code given in [78] because the framework requires actual C program code. To facilitate this requirement, the program was implemented as described. The code was then optimized to overlap the preparation for communication with the actual communication operations. This is accomplished by changing the program to communicate via immediate mode synchronous sends and immediate mode receives ($\text{MPI\_Issend}$ and $\text{MPI\_Irecv}$) coupled with $\text{MPI\_Wait}$ and then moving the message initiations as far from the completions as possible. The framework could then verify this code using the MPIC model checker using dynamic partial-order reduction, for the absence of deadlocks and the default set of assertions.

The final example requires an additional MPI procedure, namely $\text{MPI\_Irsend}$, which requires that the matching receive be posted before the “ready” mode send can be posted. The first send is forced to match the second receive using the tag field of the message. The ready mode send is posted immediately after the $\text{MPI\_Wait}$ corresponding to the $\text{MPI\_Issend}$. A second ready mode send is posted that can match only the third receive. Successful posting of the first $\text{MPI\_Irsend}$ implies that the receiver is guaranteed to be beyond that program point. Failed posting of the second $\text{MPI\_Irsend}$ implies that no guarantee can be made about further progress: thus the receiver is guaranteed to have posted the corresponding receive and no more (Figure 1.2). This verification task requires only two model processes.

### 4.2 The MPIC Intermediate Representation

The MPIC intermediate representation is a collection of C# classes that implement a transition system. At this level the program is represented by syntactic objects. There are three major groups of syntactic objects, represented by abstract classes: Transition, Operand, and Symbol.

The intermediate representation begins as a set of function graphs where a program state is a node in the graph and a transition is an edge in the graph,
operands and symbols are used to help describe transitions. The remainder of this section will describe these structures that make up a function.

### 4.2.1 Program States

Every program control location (i.e., for each PC value) there is a program state (these are referred to simply as states unless disambiguation is needed). Every state has a list of in transitions and out transitions, two PC value representations, one that corresponds to the Phoenix generated label and one that is the state number assigned to the given state by the framework (see Section 4.3.3.6). In addition, at each state, the cone of influence with respect to symbols that are branch predicates and MPI function call arguments are housed. This set of symbols is later referred to as the “set of preserved symbols” at a given state.

### 4.2.2 Program Transitions

There are seven types of transitions modeled in the MPIC IR. Each of these corresponds to one or more of the Phoenix IR statements, however the transitions in our framework are much simpler with an emphasis on simplification and modeling. Each transition has a start state and an end state.

The action modeled by the transition depends upon the individual transition type. Each transition, however uses some number of operands that represent the variables being read and written by the given transition.

#### 4.2.2.1 The EnterFunctionTransition Transition

The \texttt{EnterFunctionTransition} is a placeholder inspired by the Phoenix IR for entry to a given function. This transition knows the names of the parameters and their order in the function signature. It is used when performing a function inline operation to determine which arguments passed to the function will be assigned to the given $\alpha$-renamed parameters. All \texttt{EnterFunctionTransition} transitions are removed by the process of inlining (see Section 4.3.3.1).
4.2.2.2 The ReturnTransition Transition

The ReturnTransition is also a placeholder for the return statement of each function. If the return is implicit in the source code, Phoenix makes the return statement explicit before the MPIC IR is extracted. When inlining, this transition provides the operand that is assigned to the target of the function call. All ReturnTransition transitions are removed by the process of inlining.

4.2.2.3 The CallTransition Transition

The CallTransition represents the call site of function calls in the source code. This transition has the argument operands that are passed to the function and the operand that is the target of the return.

Each call transition is explicitly represented in the MPIC IR. MPI functions, memory allocation, and functions named assert are preserved when the graph is cleaned (see Section 4.3.3.2). The remaining function calls are passed into the TLA+ model in the form of TLA+ operators, or into the MPIC models as MPIC transitions.

4.2.2.4 The BranchTransition Transition

The branch transition represents both conditional and unconditional branches, including goto statements. When a conditional branch is needed, one BranchTransition object is used to represent the true branch, one for the false branch. In the produced TLA+ model, if the branch is conditional, the source operand is checked against zero as a guard for the true branch; the guard is negated in the false branch.

4.2.2.5 The ValueTransition Transition

The ValueTransition represents common assignment. Since the program has been simplified into single static assignment (SSA) form, there are usually more ValueTransitions in the simplified program than shown in the C source code. The left hand side of the assignment is the destination operand, the right hand side has two, or one operand(s) that could be constants or variables or some mix.
4.2.2.6 The CompareTransition Transition

The CompareTransition represents an assignment where the right hand side evaluates to a logical value.

4.2.2.7 The SwitchTransition Transition

The SwitchTransition represents a multi-way branch.

4.2.3 Transition Operands

There are five types of Operand objects: LabelOperand, MemoryOperand, FunctionOperand, VariableOperand, and ImmediateOperand. Program counters and control labels are represented using LabelOperand objects. Pointers to memory are represented using MemoryOperand objects. When an address is taken in the user program, this is represented using a MemoryOperand object that maintains a pointer to the referenced object. Functions can be passed as arguments. A place holder exists for the corresponding FunctionOperand; however it has been assumed that no function pointers are used in the input source. Variable and Immediate operands represent the syntactic objects for variables and constants as their names suggest.

4.2.4 Symbols

Each operand has an associated symbol that is used to abstractly represent the variable through the program. Symbols contain a reference to all operands that represent that symbol. Symbols are used to facilitate $\alpha$-renaming of variables when doing inlining, and to facilitate variable allocation when producing the model to be checked.

4.3 Graph Transformations

Now that a set of function graphs exists for a program, a variety of transformations can be performed, progressing towards creating a model of the given program. Most of the transformations have the same general execution structure, needing to either visit each transition in the function graph exactly one time, or needing to continue visiting transitions until a fixed point is reached. As such two strategies for graph traversal have been implemented, depth-first search, and Chaotic Iteration
based search. Each of the necessary graph transformations have been implemented using a generic visitor interface [55], with the exception of loop removal.

This section describes the implementations of the two graph traversal strategies, and the various visitors used to perform graph transformations. The loop removal heuristic is also discussed.

### 4.3.1 Forward and Reverse DFS

Simple graph traversal is handled using forward or reverse depth first search. In the algorithm to follow, it is assumed the existence of operators `push` and `pop` for stacks, `intrans` returns the array of transitions that lead into a program state, and `pre` returns the program state at the start of a transition. The `visit_trans` function modifies the transition according to the visitor that is traversing the function graph.

The initial state and the set of transitions used (either the transitions that enter the state, or transitions that exit the state) determine whether the search is forward or reverse. As such, only the algorithm for reverse post-order depth-first function graph traversal is shown in Figure 4.4. Exchanging `intrans` for `outtrans`, `pre` for `post`, and the end state of a function graph for the start state of the function graph causes this algorithm to perform a forward depth-first post-order traversal.

An auxiliary stack $l$ (the index *location*) is introduced to facilitate iterating through the transition array. The initial state and index 0 are pushed onto the respective stacks. While the state stack is nonempty, a reference to the top state is returned, along with the current index corresponding to this state. Now the number of transitions that go into the state is checked against the current index. If there are more transitions that enter this state, then the start state of this transition `pre(intrans(q)[i])` is pushed onto the state stack, along with the corresponding initial index 0 on the location stack $l$. Otherwise, the algorithm is ready to back off from this transition, visit it, and increment the location index. The natural initial state for use with the algorithm of Figure 4.4 is the end state of a function graph.
\begin{verbatim}
q   : state
s   : stack
l   : location stack
h   : state set
i   : int location
intrans : transition array
pre(t) : the start state of t

1  q := initial state
2  push(s, q)
3  push(l, 0)
4  h := ∅
5 6  while size(s) > 0
7      q := top(s)
8      i := pop(l)
9      if i < size(intrans(q))
10         if q /∈ h
11            h := h ∪ {q}
12            push(s, pre(intrans(q)[i]))
13            push(l, 0)
14         else
15            visit_trans(intrans(q)[i])
16            push(l, i + 1)
17      end if
18     else
19        pop(s)
20     end if
21  end while
\end{verbatim}

**Figure 4.4.** An algorithm for reverse depth-first program graph traversal.

### 4.3.2 An Algorithm for Chaotic Iteration-based Traversal

The chaotic iteration algorithm is a work-list algorithm based on the lattice traversal of [63]. The algorithm used is shown in Figure 4.5. This algorithm is intended to only visit transitions of the program graph.

In the algorithm to follow, the existence of operators \texttt{push} and \texttt{pop} are assumed for stacks, \texttt{intrans} returns the array of transitions that lead into a program state, and \texttt{pre} returns the program state at the start of a transition. The existence of predicate \texttt{modified}, that returns true when one of the transitions is modified by the \texttt{visit_trans} operation is assumed, along with the existence of the operation \texttt{reset} such that \texttt{modified(q) = true} exactly when \texttt{modified(reset(q)) = false}.

The work-list algorithm proceeds as follows. The work-list is initialized to contain all transitions from the function graph. While the work-list is not empty
\[ t \quad : \text{transition} \\
q \quad : \text{state} \\
W \quad : \text{work list (stack)} \\
intrans \quad : \text{transition array} \\
\text{pre}(t) \quad : \text{the start state of } t \\
\text{modified}(q) \quad : \text{predicate on } q \]

1. \( W := \text{all transitions of the function graph} \)
2. \( \text{while size}(W) > 0 \)
3. \( t := \text{pop}(W) \)
4. \( \text{visit_trans}(t) \)
5. \( q := \text{pre}(t) \)
6. \( \text{if modified}(q) \)
7. \( \text{reset}(q) \)
8. \( \text{for each } t \text{ in intrans}(q) \)
9. \( \text{push}(W, t) \)
10. \( \text{end for each} \)
11. \( \text{end if} \)
12. \( \text{end while} \)

**Figure 4.5.** An algorithm for chaotic iteration based program graph traversal.

The first transition in the list is removed from the list and visited. If the visiting of the transition \( t \) modified the start state, \( \text{pre}(t) \), then all of the transitions that go into \( \text{pre}(t) \) are added to the work-list \( W \).

### 4.3.3 Visitors

The sequence of visitors [55] that are applied to a given function graph to accomplish the transformations required is presented in this section. Each visitor retains a reference to the function graph being visited. The visitors implemented provide the following additional functionality on function graphs:

- Inlining a function graph into all corresponding call transitions of that function in another function graph.

- Creating a copy of a function graph.

- Cleaning a function graph.

- Computing the cone of influence for control variables and MPI operation arguments in a function graph.

- Eliminating no-op transitions in a function graph.
• Numbering function graph states.

• Locating status variable references in a function graph.

4.3.3.1 Inlining of Functions

The inlining of functions is handled using a fixed point based iteration. A function is repeatedly inlined over the remaining collection of functions until all call sites of the function are removed from the program. It is assumed that there are no recursive function calls and no function pointers used in the program being modeled. It is further assumed that all function parameters are value parameters.

Individual replacement of a call transition with a function body uses the visitor with the depth first search of Section 4.3.1. A reference to the function graph \( f \) being inlined is retained in the visitor. Whenever a call transition referencing \( f \) is located in the function being traversed by the visitor, the call transition is removed, the body of \( f \) is copied, the variables of \( f \) are \( \alpha \)-renamed, the new copy of \( f \) is inserted by creating a sequence of ValueTransition transitions that assign the values of the call arguments to the \( \alpha \)-renamed parameters of \( f \), and if there is a source operand in the return statement, the return transition of \( f \) is replaced by a value transition assigning the returned operand to the target of the initial function call, otherwise an unconditional branch to the post state of the call transition is used.

Copying the body of a function graph also uses a depth first search visitor. When a transition is visited, a copy of the start and end states are created, if necessary, and given unique labels (the state labels must also be \( \alpha \)-renamed since the same function could be inlined at multiple function call sites). Each transition is cloned and registered with the new function body. Cloning the transition causes all of the operands of the transition to be copied. It is important to point out that the \( \alpha \)-renaming of the function’s variables takes place after the function body is copied — preserving the original function graph.
4.3.3.2 Cleaning the Function Graph

Once all of the functions defined in the program text have been inlined there remains a single function graph to represent the program. Remaining function call transitions are one of four possibilities:

1. The call is an MPI procedure.
2. The call is a memory allocation.
3. The call is an assertion (if the user desires these to be preserved).
4. The call is some other library routine that is unknown to the model checker back-ends.

In the event any calls that are not known to the model checking back-ends remain, they must be removed in order for model checking analysis to proceed. Again, using the DFS visit of Section 4.3.1, each transition is checked. If the transition is a CallTransition that falls into the fourth category, it is changed into an unconditional branch (i.e., a goto) that is effectively a no-op. Unconditional branches are removed later (see Section 4.3.3.4).

Clearly if the user writes an assertion about the return value or side-effect of some library call that is not defined in the user’s source code, the model checkers will not be able to reason about that assertion.

4.3.3.3 Creating a Control Skeleton via Slicing

Creating a slice of the function graph requires some way of determining which transitions to keep and which to throw away. To preserve deadlocks, local assertions, and the presence of cycles, the cone of influence is computed for variables that are (i) branch predicate operands, (ii) operands used as arguments in MPI function calls, and (iii) operands in assert transitions. The slice is then computed based on these cones.

To compute the cone of influence, the Chaotic Iteration algorithm described in Section 4.3.2 is used. The visitor used examines the symbols corresponding to the
operands of the transitions based on the transition type. For all transition types, the union of the preserved set of symbols at the end state and the preserved set of symbols at the start state is assigned to the preserved set of symbols at the start state.

If the transition is an MPI function call, an assertion, or a branch, then the symbols corresponding to the source operands are added to the preserved symbol set at the start state and the start state is considered modified. For other transition types, if the transition writes a variable in the preserved symbol set at the end state, then the symbols corresponding to the operands that are read by the transition are added to the preserved symbol set at the start state which is then considered modified. If the symbol corresponding to the operand written by the transition does not also correspond to an operand that is read in the same transition then it is removed from the preserved symbol set at the start state.

To compute the slice of the program, depth-first visit of Section 4.3.1 is used. MPI procedures, branches, and assertions are ignored. For all other transitions, if the symbol corresponding to the operand written by the transition is in the preserved symbol set at the transition end state then the transition is preserved. Otherwise the transition is changed into an unconditional branch (a no-op).

4.3.3.4 Eliminating No-op Transitions

No-op removal is performed using the forward variant of the algorithm described in Section 4.3.1. The visitor ignores all transitions that are not branches. Unconditional branches represent no-op transitions in the model.

To remove unconditional branch transitions, the end state is removed from the function graph, as is the transition. The transition is removed from the in transition list held at the end state. Each out transition from the end state is added to the out transition set of the start state. Each in transition held at the end state is then added to the list of in transitions at the start state. The unconditional branch being removed can then be removed from the in transition list at the start state.

In the process of this no-op elimination, it may become the case that both transitions of a branch may share the same end state — this is also a no-op. In
In this case it becomes acceptable to remove both arms of the branch. The algorithm is as above except both transitions are removed from the function graph.

4.3.3.5 Removing Empty Counting Loops

One consequence of the above simplifications is that all loops (not necessarily loop bodies) are preserved in the simplified program graph. When the loop is preserved but the corresponding loop body has been removed then it is desirable to also remove the loop.

Because the transformation of the original C program may introduce temporary variables, not all of the remaining empty loops have the same form. Some use temporary variables as part of the increment of the branch condition variable, while others increment the branch condition variable directly. The presence of this additional transition, and not knowing precisely which variable is the branch condition variable, causes us to propose the following heuristic that does not use the visitor structure above, but an approach with more of a model checking flavor:

Visit the graph in forward depth first search order. When a state is revisited (i.e., there is a loop in the function graph), if it is a cycle of length four or five, then it is assumed to be an empty counting loop. The conditional branch transitions are then identified. Each transition, including the branch that enters the loop, to the program state that is revisited are removed from the function graph. The branch transition that does not enter the loop is changed into an assignment based on the loop bound. Thus if the loop condition was \( i < 100 \), then a value transition having the assignment \( i = 100 \) is put in the place of the exit branch.

4.3.3.6 Numbering Program States

The MPIC tool (described in Chapter 5) requires an integer PC value for program states. A visitor is created to use the algorithm of Section 4.3.1 to visit the remaining states in the function graph and number them appropriately.
4.3.3.7 Finding MPI Status Variables

The Microsoft compiler front end flattens all data structure access into a base address and offset. This means user defined type information is not present in the meta-data of the Phoenix IR. The status variables must take on certain values depending on the outcome of the applied MPI operation under the MPI standard. To allow for this multi-field structure those status variables that may exist in the user program must be located.

To facilitate the discovery of the status variables, the framework visits the graph using the depth-first search of Section 4.3.1. Whenever a call transition is discovered, if it is also an MPI function call, the operand location of the status is determined using a table lookup. The symbol corresponding to the operand in the function graph is then re-typed to reflect that it is in fact a status struct.

This typing information is used when creating the TLA+ model of an MPI program. When a status variable is discovered, the appropriate number of memory locations in the model are generated and the corresponding addressing of model symbols is incremented.

4.4 Producing Models

After the desired simplifications have been performed on the function graph, a model can be created for model checking based analysis. To facilitate this analysis, it must be the case that the model is minimally inlined (see Section 4.3.3.1) and cleaned (see Section 4.3.3.2). The framework can output models in a choice of three languages: TLA+ and MPIC for model checking based analysis, and DOT [21, 22] primarily for debugging.

4.4.1 TLA+

This section shows the intended template for producing a TLA+ model in Figure 4.3. This template represents the desired final formatting of the program transition relation. The relation is produced by iterating over the transitions in the function graph, causing each to format itself into the TLA+ notation.
Before the transition relation, there are three major pieces of information that
must be inserted: the initial state of Memory, the mapping of variable names onto
memory addresses, and the corresponding label for the final program state of the
function. The TLA+ formatted output for the model of Figure 1.2 is contained in
Appendix B.

### 4.4.2 MPIC

The MPIC model checker is more tightly integrated with the model extraction
framework. In particular, it has no parsing facility — rather it requires the tran-
sition relation and variable information to be composed into a data structure and
passed, either on disk, or in memory, to the model checking tool.

The transition relation is created using a sequence of passes over the function
graph. The first pass numbers states (see Section 4.3.3.6). The second pass com-
bines conditional branch transitions to accommodate the model checker’s transition
structure (see Section 5.2.3). Transitions are then sorted by the integer PC value of
the transition start state. The model checker’s corresponding transition structure is
created for each transition type in the model and inserted into the transition array.

Due to the limited number of operations that the MPIC model checker handles,
some simplifications are made at this stage. The MPI operations MPI_Init and
MPI_Finalize are replaced by no-ops. The MPI_Comm_rank and MPI_Comm_size
operations are replaced with assignments to predefined values pid, and N respect-
ively. The variables pid and N for each process are initialized by the MPIC model
checker. Other MPI operations that are unknown to MPIC are also replaced with
no-op transitions.

### 4.4.3 DOT

The DOT [21, 22] output has been used primarily for debugging the function
graphs and the graph transformations described above. Each model transition
produces an edge in the graph image; program states produce nodes. Depending
upon the desired verbosity, the image can include the transition text, text labels,
and cone of influence information at a given state.
4.5 Related Work

A number of frameworks exist that create models automatically from C source for model checking based analysis. In this section a brief overview of the major platforms currently available is given along with a comparison of their features.

The CBMC model checker [14] handles all ANSI C constructs and performs a sat based bounded model check [12] of a finite unrolling of the program. Error trails are replayed using a debugger like environment. Concurrency is not handled, neither are libraries.

The Fsoft [40] framework compiles C programs into circuit modules using CIL [62] as a front end. This is extended to handle threads [41]. It is not known whether other libraries or methods of concurrency are handled. Fsoft uses the DiVer [26] model checker as a back-end.


The Verisoft tool [31] drives the program code directly, controlling the scheduling. Here assumptions are made regarding the presence of cycles. In particular the search depth is limited or the driver code generates a finite number of external events.

The Modex/FeaVer framework [37, 38] is a tool to extract and verify features of ANSI-C programs against a user provided specification, using the SPIN model checker as the target back-end for the verification tasks.

The MPIC verification environment integrates with the Visual Studio IDE and provides the user a simplified interface for the application of formal verification tools, similar in spirit to each of the above listed frameworks. This framework makes extensive use of the Microsoft Phoenix research compiler framework to accomplish default program simplification. Thus details such as parsing C programs and simplifying them into a homogeneous single static assignment format is handled by the Phoenix compiler before performing any analysis. Similar to the Fsoft
framework slicing is used to reduce the size of the models being verified. However, unlike previous work, this framework does an approximate slice that preserves the communication skeleton while discarding most data — thereby preserving reactive properties such as deadlocks, local user placed assertions, the presence of cycles, and a set of provided default assertions. A second way this framework is unlike related work is that error trail simulation drives the actual distributed program using the Microsoft Visual Studio remote debugger. Thus when an error is found, the trail, values of the variables, and execution interleaving order can all be examined in the familiar Visual Studio IDE.

4.6 Summary

This chapter has presented a modeling framework that extracts and simplifies models of MPI-based programs written in C. The framework uses the Microsoft Phoenix compiler as a front end and inserts a compilation pass to read the Phoenix intermediate representation and create a separate intermediate representation. Using this separate intermediate representation, the MPIC IR, program models are simplified by inlining functions and slicing the resultant function graph to preserve the communication skeleton of the program. Program models can then be produced in TLA+ or MPIC for model checking based analysis or DOT for debugging.
CHAPTER 5

IMPLEMENTATION OF THE MPIC
MODEL CHECKER

The model checking analysis available in the form of the TLC model checker has some severe limitations. Due to the exponential complexity of model checking asynchronous processes, the size of the state space of even simple programs having larger numbers of processes is beyond the reach of model checker tools that do not have some form of partial-order reduction. The TLC model checker has no form of partial-order reduction.

A second problem with TLC is as follows: The overhead of computing the next state in TLC gets quite large for more than a few processes. This is primarily due to the fact that TLC does not restrict the model checking to an interleaved product, although for an asynchronous program verification that preserves local assertions, deadlocks, and the presence of cycles, an interleaved product is all that is required. In addition the next state of each variable in the model is computed at every transition — however in an asynchronous program model very few variables are written in each transition. Thus there is significant time overhead when using TLC for model checking MPI programs due to the great generality of the TLC tool.

Chapter 3 presented a dynamic partial-order reduction algorithm for a subset of the MPI operations, namely MPI_Isend, MPI_Irecv, MPI_Wait, MPI_Test, and MPI_Barrier. These semantics are derived from the model of Appendix A. This chapter presents a model checker prototype called MPIC (MPI Checker) that implements this subset of operations and the dynamic partial-order reduction algorithm. In addition this chapter discusses the design decisions that were made to make it possible for MPIC to handle larger numbers of processes in a given model.
The remainder of this chapter is organized as follows. A set of working assumptions are posed in Section 5.1. Section 5.2 describes the high level architecture of MPIC, the structure of the state vector, the mechanics of transitions, and how expressions are evaluated. Section 5.3 describes the four variations on the standard depth first model checking algorithm that are implemented in MPIC. Optimizations that facilitate larger numbers of processes are discussed in Section 5.4. Section 5.5 details how and when the various properties are checked using MPIC and how a violation is reported. Section 5.6 gives an overview of related work. Section 5.7 summarizes the contributions of this chapter.

5.1 Assumptions

The MPIC model checker is a domain specific explicit state enumeration tool. It is intended for use with programs written in the Single Program Multiple Data (SPMD) style. As such it takes as input only one transition relation and one variable set and instantiates \( N \) processes each having a copy of the transition relation and variable set. The MPI operations are considered atomic and assumed to be side-effect free. MPI data structures such as requests are also considered opaque.

It is assumed that there is only one function in the program, main, and that there are no library calls except for MPI operations, assert statements, and memory allocation. Thus it is assumed that the model has been \emph{inlined} (Section 4.3.3.1) and \emph{cleaned} (Section 4.3.3.2), at the least, before the model checking takes place.

The following assumptions are made with regards to the properties that will be checked with the MPIC tool. The common theme in this dissertation is the preservation of deadlocks, local assertions, a set of default assertions, and the presence of cycles. As such, MPIC does preserve these properties.

5.2 Architecture

MPIC is organized to be modular and extensible. This section covers the primary data structures contained within MPIC: state vectors, the search stack, transitions, and expressions.
5.2.1 State Vectors

The model checker has at most one state vector that represents the current values of the variables of all processes in the model. The state vector is divided into a local and global store for each process. The local store contains the PC value for each process and represents the memory available to that particular process.

The global store of a process contains the request sequence that is created by executing the available MPI operations. The global store also contains the collective context for implementing the `MPI_Barrier` construct. The global store is accessible only via MPI operations.

Memory to house the local store of a process and message requests is allocated upon request by the model, except for those variables that are represented by MPIC IR symbols when the model is generated. Each of the MPIC IR symbols is allocated a memory space and address when the model checking procedure is initialized.

5.2.2 The Search Stack

A static handle to the state vector is contained in the state class. This class contains all of the additional information that is needed to continue the model checking procedure. This information includes (i) which processes have already tried moved, (ii) which potential message matches are available, (iii) the Undo structure (see Section 5.2.3) for the transition that executed to generate this instance of the state class, and (iv) whether any process has successfully generated a new state from this state. This information remains on the stack for each state, each state having a reference to the global state vector.

5.2.3 Transitions, Forward and Backward

The transition relation is represented using an array of transition objects. This array is indexed by the current PC value of the executing process. When a transition is executed it modifies the local store and may modify the global store for the given process. Under certain circumstances the local store and global store of a remote process participating in a communication will also be modified.

All transitions create an Undo object that corresponds to the transition type
being executed. The Undo object is retained in the search stack until the state
generated by the transition that generated the Undo is popped from the search
stack. At this point the Undo is applied to the state vector, reversing the effect of
the transition that executed to generate the state.

Eight transition classes are implemented that correspond with the semantic
definitions of Chapter 3 and their associated Undo objects. In addition to modi-
fying the state vector, a transition can determine when it is enabled, and provide
information on which variable changed as a result of executing the given transition.

The individual transition types are

- **Assignment**: This transition type is used where an assignment to a variable
  in the local store is made. Assignment is used to model both the value
  transition and compare transition of the MPIC IR described in Sections 4.2.2.5
  and 4.2.2.6 respectively.

- **Goto**: All branch types are modeled using a conditional goto. This transition
  evaluates an expression and assigns either the evaluated PC expression of the
  corresponding true or false branch to the PC variable of the executing process.

- **Assert**: The assert statement evaluates its argument. In the style of C, if the
  argument evaluates to zero an error is reported; otherwise, the assert has no
  effect except to increment the PC of the calling process.

- **MPI_Issend**: This operation increments the PC and updates the global store
  of the calling process by posting a send request.

- **MPI_Irecv**: This operation increments the PC and updates the global store
  of the calling process by posting a receive request.

- **MPI_Wait**: This operation is disabled until the communication specified by the
  corresponding request has completed. When enabled, this operation updates
  the PC of the calling process and transmits values from sender to receiver if
  the calling process is the first to call MPI_Wait or MPI_Test. Data are also
  transmitted if needed.
- **MPI_Test**: The MPI_Test does not block, and is therefore always enabled. If possible the MPI_Test completes the communication specified by the associated request. If it can complete the communication it writes a nonzero value into the flag variable’s memory location. Otherwise a zero is written. In either case, the PC of the calling process is also updated. Data are also transmitted if needed.

- **BarrierInit**: The MPI_BARRIER is entered using this transition. The protocol is exactly as described in Section 2.11. Aside from the PC on the calling process and the collective context, no other state is updated.

- **BarrierWait**: The MPI_BARRIER is exited using this transition. The protocol is exactly as described in Section 2.11. Aside from the PC on the calling process and the collective context, no other state is updated.

Undo objects for each of the above transitions are created as the transition is executed. Each of the variable locations that are written by the transition is saved by the Undo before they are written.

### 5.2.4 Expressions

Four types of expressions exist in the current implementation. Each is concrete, and based on integer values. The expressions available are variable, constant, binary, and unary forms. The C convention for Boolean values, namely zero is considered false and nonzero values are considered true is used.

The polymorphism mechanism of the .Net runtime is used to implement an expression evaluator. The individual arithmetic and logical operations are implemented using the C# operators.

### 5.3 Variations on Depth First Search

The primary algorithm for model checking in MPIC is the depth first search (DFS). Four variants of DFS are implemented to provide the options of dynamic partial-order reduction, stateless search, both optimizations, or neither.
Stateless search is perhaps the simplest form of model checking. Shown in Figure 5.1, this algorithm performs the full expansion of each state using two auxiliary functions that are described using natural language, but should be fairly straightforward.

The enabled function returns the set of transitions that are enabled at a given state \( q \). When no transition is enabled in \( q \), the set is empty. The nextenabled function iterates through the transition set returned by enabled such that each transition is returned only once at a given state.

This search preserves only local user placed assertions, default assertions, and deadlocks. It cannot handle a state space graph that contains a cycle. The search not only visits every state, but may in fact visit large portions of the state space repeatedly.

To add dynamic partial order reduction and a state table, the algorithm of Figure 5.1 is modified to that of Figure 3.12. The proof of correctness for this algorithm is similar to that of Theorem 5 in Chapter 3. The primary differences are that this algorithm does not detect cycles and does not use stateful search.

For performance reasons cycle detection is only implemented in connection with the stateful search. The variants implemented in the MPIC tool are (i) stateless

\[
q, q' : \text{state} \\
\text{s} : \text{stack} \\
\text{1. } q := \text{initial state} \\
\text{2. } \text{push(s, q)} \\
\text{3. } \\
\text{4. while size(s) > 0} \\
\text{5. } q := \text{top(s)} \\
\text{6. (⋆ Deadlocks are checked here. ⋆)} \\
\text{7. if } \exists t_p \in \text{enabled}(q) : \\
\text{8. } t_p := \text{nextenabled}(q) \\
\text{9. (⋆ Local assertions are checked here. ⋆)} \\
\text{10. } q' := t_p(q) \\
\text{11. } \text{push(s, q')} \\
\text{12. else} \\
\text{13. } \text{pop(s)} \\
\text{14. end if} \\
\text{15. end while} \\
\]

**Figure 5.1.** An algorithm for stateless depth first search based model checking.
search as shown in Figure 5.1, (ii) stateless search with dynamic partial-order reduction as shown in Figure 3.12, (iii) stateful search with cycle detection, and (iv) stateful search having cycle detection and dynamic partial-order reduction as shown in Figure 3.11.

5.4 Optimizations

Model checking is compute bound and memory bound, requiring attention to be placed on optimizing the implementation. MPIC has three optimizations that help mitigate the computational costs: deltas, incremental hashing and allocation, and incremental error trails.

5.4.1 Deltas

State deltas are a technique to reduce the cost and memory overhead of model checking by creating only one complete working state vector. When a forward transition is executed the working state is updated. When the Undo is applied, those changes caused by the corresponding forward transition are reversed.

5.4.2 Incremental Hashing and Allocation

Memory is allocated as needed by the requesting process. The MPIC model checker uses an incremental hashing based on the Stern and Dill algorithm of [84]. The hashing algorithm needs three random integers for every bit in a given state vector to compute the hash. These integers are allocated on demand while performing the model checking.

Individual memory cells are aware of their contents and are able to hash them when the contents change. The hash value of the old memory contents is removed from the composite hash, and the new hash value is introduced (both use the xor operation).

5.4.3 Incremental Error Trails

One surprise in developing MPIC was the overhead of large error trails caused by the presence of many concurrent processes. The solution developed is an incremental error trail. When an error is reported by the model checker, the stack trace
(which can be quite long) is not produced. Instead, the model checking procedure stops and the working state is reset to its initial values.

A pointer to the search stack is passed to the error trail handling routine, retaining the search stack in memory. The model checker object can then be instructed to move forward one step by executing the transition on the stack and report the resulting change in the state vector.

5.5 Properties Checked by the MPIC Model Checker

The MPIC tool checks for assertion, cycle, invalid end state, invalid rank, and invalid request violations. When a violation is discovered an exception is thrown by the model checker which halts execution. The exception is caught by the model checker just above the main model checking algorithm.

5.5.1 Assertion Violations

Assertions are user placed statements that require an expression. Assert transitions are always executable. If the expression evaluates to zero when the assert transition is executed, the transition throws an exception.

5.5.2 Cycle Violations

Cycles are discovered in the model checking algorithm of Figure 3.11 on line 17. Since searching the stack would be quite expensive, a bit is placed in the hash table to indicate whether the state being visited is in fact on the search stack. Thus it is the hash table that throws the exception, stopping execution and transferring control to the error trail setup.

5.5.3 Invalid End State Violations

An invalid end state is a state where there are no enabled transitions and the state is not the final state in the model. When the state is evaluated to determine whether there are any transitions in enabled, an exception is thrown if the above criteria are met.
5.5.4 Invalid Rank Violations

Rank violations happen when a process tries to communicate with a rank that does not exist in the model. This is checked when the set of possible message matches at a given state are computed.

5.5.5 Invalid Request Index Violations

If the request index referenced by an MPI_Wait or MPI_Test operation is out of bounds, an invalid request index exception will be thrown. The MPIC model checker does not deallocate requests, so an out of bounds request handle would be one that has never been allocated along the current execution trace in the state space.

5.6 Related Work

Many explicit state model checkers have been developed in recent years. Perhaps the most closely related are Zing [3], SPIN [36], and PV [60]. The Zing model checker is a tool for checking local safety properties of object oriented program models. It implements both deltas and incremental hashing and is similar in those respects to MPIC. The Zing model checker, however, has a significantly more expressive object oriented language and many additional features such as incremental heap canonicalization for thread symmetry [59] that are not present in MPIC. The MPIC tool, however, has no a priori bounds on the size of state vectors or the number of processes that can be included in a given run. In addition, Zing uses a thread based model of concurrency, while MPIC uses a process model. The partial-order reduction algorithms of MPIC and Zing also differ significantly as Zing’s reduction is based on transactions [49].

SPIN [36] and PV [60] are both model checkers for Promela, offering no native library support, or custom partial order reduction for MPI.

Perhaps the most important difference between MPIC and each of these tools is the presence and availability of a selection of library routines that are built into the model checker directly and that the partial-order reduction of MPIC knows how to leverage the semantics of these library operations to effect greater scalability.
5.7 Summary

This chapter has presented an overview of the design and implementation of the MPIC model checker. The primary architectural features of MPIC are an object oriented design and implementation of transitions, expressions, and a dynamic state structure. The set of transitions and expressions allowed by the model checker can be extended in a natural way to include new and additional MPI primitives and datatypes such as floating point numbers.

MPIC implements an MPI aware dynamic partial-order reduction algorithm. In addition MPIC uses state deltas, dynamic memory, dynamic hashing allocation, and incremental hashing to both reduce the start up overhead and remove limits on the number of process that can participate in the model checking analysis.
CHAPTER 6

CONCLUSIONS

To help reason about programs that use MPI for communication, a formal TLA+ model of the point-to-point communication operations of the MPI 1.1 standard has been developed. This model serves to augment the existing natural language document. This formal specification is described, as well as a framework to extract models from SPMD-style C programs. The framework incorporates high level formal specifications, and yet allows designers to experiment with these specifications, using model checking, in a familiar developing and debugging environment. This effort has helped identify a number of omissions in the MPI reference standard document. The experience gained so far suggests that a formal semantic definition and exploration approach as described here must accompany every future effort in creating parallel and distributed programming libraries.

A domain specific partial-order reduction algorithm has been developed and theorems have been reasoned about with regards to the independence characteristics of a subset of the MPI operations. It has been shown that, under the assumptions made in this dissertation, it is not necessary to interleave the execution of entering or exiting calls to `MPI_Barrier`. It has also been shown that a process’s actions are independent with respect to actions of other processes unless those two actions satisfy the predicate `Complete`. The proposed subset of operations and the corresponding dynamic partial-order reduction algorithm have been implemented in a new model checker prototype called MPIC.

A model extraction framework has been created that abstracts away much of the computation performed while preserving properties of interest, namely user placed assertions, the absence of deadlock, the absence of cycles, and a set of default
assertions. This framework can be used with either the TLC model checker or the MPIC model checker to verify models that are automatically created based on the user’s C code.

It is believed that the approach of formalizing communication libraries and building formal semantics based partial-order reduction algorithms and verification frameworks such as this one may emerge to be a viable approach in analyzing other library based parallel and distributed programs.

6.1 Summary of Dissertation

The dissertation covers the semantics model of MPI written in TLA+ in great detail. A subset of the operations are selected and the semantics with respect to those operations is closed. Independence characteristics are then determined for the operations included in the subset.

With independence determined a dynamic partial-order reduction algorithm can be used to increase the scalability of model checking. An algorithm for dynamic partial-order reduction is proposed and proofs offered, showing that the algorithm does in fact preserve the properties of interest. An implementation of the execution semantics of this subset of MPI operations and the dynamic partial-order reduction algorithm in the MPIC model checker tool.

The end to end user experience is addressed in that a model is extracted from the user’s C code. A number of abstraction based model simplifications are provided that improve the performance of the model checking. The model can the be checked using either TLC or MPIC. If an error trail is produced by either tool, the trail is replayed interactively using the visual studio debugger environment.

6.2 Review of Contributions

The contributions of this dissertation include the following:

• An application of formal verification to a new domain — namely high performance computing software.

• The first use of library semantics in the formal verification of software.
• A formal model of the communication semantics of all of the point to point operations of MPI.

• Independence theorems about a number of MPI operations.

• A new dynamic partial-order reduction algorithm that is tailored to the semantics of MPI and the proof of correctness of this algorithm.

• A new model checker that implements a the subset of the communication semantics and dynamic partial-order reduction for MPI described in Chapter 3.

• A model extraction framework to generate and simplify models of MPI-based SPMD C programs.

6.3 Future Directions

This section attempts to detail a few ideas that have been discussed in the process of performing the research and engineering work that have been reported in this dissertation. Each of which would be very interesting future work. The section is divided into three parts dealing with the model in Section 6.3.1, the modeling framework in Section 6.3.2, and the model checker in Section 6.3.3.

6.3.1 Future Directions for Modeling MPI

6.3.1.1 Extending the TLA+ Model

Although this dissertation presents the semantics of the point to point operations of MPI, there are many MPI operations that remain to be modeled. The most straightforward extension of the existing MPI model is to add operations on groups and communicators, the hooks for which are already mostly in place.

A second, fairly simple extension is the addition of threads in the model. In Appendix C the individual processes are chosen using existential quantification. Clearly threads could be similarly scheduled.

Future plans also include overcoming the limitations of the current framework in terms of handling communication topologies. Another area where formal semantic definitions can help is in extensions of MPI to support different levels of threading.
As pointed out in [33], even short MPI programs which employ threading can have nasty corner cases. Formal specifications, as well as direct execution methods for these specifications can have maximal impact in these areas, in that capacity will not be limited in terms of model checking, and yet be able to shed light on the semantic intricacies, and pitfalls to avoid.

6.3.1.2 Using the Model

An additional direction for future research is model based testing [74]. Using the model and model checker, one could determine which points in the code are reachable, whether assertions pass or fail under some execution interleaving order, and then generate from the counter example an appropriate test harness to determine whether the program could actually be driven to that point.

6.3.2 Future Directions for the Model Extraction Framework

6.3.2.1 Interpolant-based Model Checking

The TLA model of a MPI-based C program is basically a large first order logic formula that represents a transition system. The modeling framework traverses the program graph and outputs the dynamic part of the formula, while other parts are static.

Interpolant based model checking [53] requires a logic formula to be emitted that is created by essentially a depth first traversal of a program graph. The primary issue that needs to be addressed is how to cause the modeled processes to be interleaved. The answer here should be something like the dynamic partial-order reduction developed in Chapter 3.

6.3.2.2 Language Feature Extraction

Another way that the model extraction framework could be extended involves preserving the specified language structures (structs, classes, etc.) in such a way that the model checker could take advantage of the program organization. Perhaps more simply, the addition of arrays, structs, classes, and a higher level notion of pointers would help in the modeling of a more general class of C programs.
6.3.2.3 Model Feature Extraction

One problem with the TLA+ model of MPI is the required generality of the model to handle the possible interaction from any of the other MPI operations that may be used. With a closed semantic model for the subset of MPI operations, the semantics were significantly simplified. For example, the Transmit and Pair rules of the model in Appendix A are both incorporated into the rules for Wait and Test of Chapter 3 because the language features requiring these rules were not present.

The model extractor could be extended to recognize which language features are used by a given program and select from a set of models that most closely fits the required feature set. In this way, state explosion due to un-necessary language feature interaction could be mitigated.

6.3.2.4 Abstraction Refinement

It is unclear how to abstract a communication operation and how refinement of such an operation would result. Otherwise regular counter-example guided abstraction refinement would be a nice addition to the existing abstraction techniques.

6.3.3 Future Directions for Model Checking MPI Programs

The model checker MPIC stands to gain the most from future work.

6.3.3.1 Independence of MPI Operations

There is immense potential for additional research in this area. There are many more MPI operations that require a formal semantic characterization, to show independence. It is conjectured that these proofs are mechanizable, perhaps using symbolic model checking. Once the independence theorems are known they can be used in the MPIC model checker to eliminate unnecessary redundant exploration using the dynamic partial-order reduction of Chapter 3.

6.3.3.2 Implementing Additional MPI Operations

The object oriented design of the MPIC tool makes adding new MPI operations a matter of writing a new transition class and a corresponding new Undo class. In
addition, other data-types could be introduced, allowing for arrays of floating point numbers and so on.

6.3.3.3 Symmetry for MPI

Symmetry reduction seeks to reduce the number of states visited by creating equivalence classes of states under some permutation of the state vector. Most of the programs that use MPI for communication are written in the SPMD style. This means all of the processes use the same program image. Now the question is how does one define symmetry over a set of identical processes where all are equivalent except for the rank of the given process? One, perhaps simplistic answer is to sort the processes, ignoring their rank.
APPENDIX A

THE TLA+ MODEL OF THE MPI 1.1 STANDARD

The following model is a mathematical representation of the communication semantics of MPI as described in [82]. Although we have carefully created this model, we make no guarantees with regards to correctness.
The formal MPI library specification.

Robert Palmer
The University of Utah
School of Computing

EXTENDS Naturals, TLC, Sequences, FiniteSets

Constants are given values in the configuration file that accompanies this document: mcc.cfg corresponds to the MSMPI implementation

CONSTANTS

$N$, The number of processes in the computation.

$MAX\_COMM$, The highest allowed handle value for a communicator. This is not in the standard but makes our model finite.

$MAX\_GROUP$, The highest allowed handle value for a group.

$RANK\_ORDERINGS\_SIGNIFICANT$, A flag to indicate whether all possible ranking orders should be considered in verification.

Return Codes

$MPI\_SUCCESS$, The return value of a successful call to an MPI procedure.

$MPI\_ERR\_BUFFER, 
MPI\_ERR\_COUNT,
MPI\_ERR\_TYPE,
MPI\_ERR\_TAG,
MPI\_ERR\_COMM,
MPI\_ERR\_RANK,
MPI\_ERR\_REQUEST,
MPI\_ERR\_ROOT,
MPI\_ERR\_GROUP,
MPI\_ERR\_OP,
MPI\_ERR\_TOPOLOGY,
MPI\_ERR\_DIMS,
MPI\_ERR\_ARG,
MPI\_ERR\_UNKNOWN,
MPI\_ERR\_TRUNCATE,
MPI\_ERR\_OTHER,
MPI\_ERR\_INTERN,
MPI\_ERR\_PENDING,
MPI\_ERR\_IN\_STATUS,
Assorted Constants

- **MPI_BOTTOM**: Section 3.11 Null Processes
- **MPI_PROC_NULL**: The wildcard source rank.
- **MPI_ANY_SOURCE**: The wildcard tag value.
- **MPI_UNDEFINED**: A special rank returned to a process that is not a member of the queried communicator.
- **MPI_BSEND_OVERHEAD**, **MPI_KEYVAL_INVALID**, **MPI_ERRORS_ARE_FATAL**, **MPI_ERRORS_RETURN**

Maximum sizes for strings

- **MPI_MAX_PROCESSOR_NAME**, **MPI_MAX_ERROR_STRING**

Elementary Datatypes

- **MPI_CHAR**, **MPI_SHORT**, **MPI_INT**, **MPI_LONG**, **MPI_UNSIGNED_CHAR**, **MPI_UNSIGNED_SHORT**, **MPI_UNSIGNED**, **MPI_UNSIGNED_LONG**, **MPI_FLOAT**, **MPI_DOUBLE**, **MPI_LONG_DOUBLE**, **MPI_BYTE**, **MPI_PACKED**

Datatypes for reduction functions

- **MPI_FLOAT_INT**, **MPI_DOUBLE_INT**, **MPI_LONG_INT**, **MPI_2INT**, **MPI_SHORT_INT**, **MPI_LONG_DOUBLE_INT**

Special datatypes for constructing derived datatypes

- **MPI_UB**, **MPI_LB**
Reserved communicators

- **MPI_COMM_WORLD**: The handle for `MPI_COMM_WORLD`.
- **MPI_COMM_SELF**: The handle for `MPI_COMM_SELF`.

Results of communicator and group comparisons

- **MPI_IDENT**: 5.4: Two communicator handles refer to the same communicator.
- **MPI_CONGRUENT**: The communicator handles are different; communicators differ only in context.
- **MPI_SIMILAR**: The communicator handles are different; communicators have the same group, however both context and ranking differ.
- **MPI_UNEQUAL**: The communicator handles are different; communicators have different groups, contexts, and rankings.

Environmental inquiry keys

- **MPI_TAG_UB**: The upper bound on the tag range 19.27 – 19.31
- **MPI_IO**
- **MPI_HOST**
- **MPI_WTIME_IS_GLOBAL**

Collective operations

- **MPI_MAX**
- **MPI_MIN**
- **MPI_SUM**
- **MPI_PROD**
- **MPI_MAXLOC**
- **MPI_MINLOC**
- **MPI_BAND**
- **MPI_BOR**
- **MPI_BXOR**
- **MPI_LAND**
- **MPI_LOR**
- **MPI_LXOR**

Null handles

- **MPI_GROUP_NULL**
- **MPI_COMM_NULL**
- **MPI_DATATYPE_NULL**
- **MPI_REQUEST_NULL**: A special handle value for requests. Set this to 0 in the configuration file and make the initial values of the requests...
occupied to avoid an array out-of-bounds error.

MPI_OP_NULL,
MPI_ERRHANDLER_NULL,

Empty group
MPI_GROUP_EMPTY,

Topologies
MPI_GRAPH,
MPI_CART

Variables represent the state of the MPI system at any given time. None of these state elements are specified by the standard. However they are useful to describe what is specified. In particular mention is made of handles that reference opaque objects. The communicator and requests arrays are such opaque objects that are referenced by integer handles that in our model are unique across both space and time (i.e., the same value is used for MPI_COMM_WORLD on all processes for the entire execution etc.).

VARIABLES

communicator, An array of communication universe objects.
bufsize, The size of the user attached message_buffer.
message_buffer, The user attached buffer.
requests, A array of message requests lists, one per process. Although we do model the allocation of request objects by adding a structure to a list of requests, we are not modeling the freeing of requests more than setting the associated handle to MPI_REQUEST_NULL.
initialized, An array of flags that indicate whether MPI_Init has been called by a given process.
collective, The collective contexts for all communicators
group, The array of groups
Memory A model of memory for individual processes.

Type invariants are used to enforce a typing system for variables in the model being verified. The invariants to be checked are specified in the corresponding configuration file.

Memory is considered a program_var

mpi_vars \triangleq (group, communicator, bufsize, message_buffer,
requests, initialized, collective)

Messages \triangleq 
[src] : (0 .. (N − 1)) \cup \{MPI_ANY_SOURCE,
\[\text{MPI}\_\text{PROC}\_\text{NULL}\}, \quad 21.24 - 21.25\]

\[
\begin{align*}
\text{dest} & : (0 .. (N - 1)) \cup \{\text{MPI}\_\text{PROC}\_\text{NULL}\}, \quad 19.39 \\
\text{msgtag} & : 0 .. \text{MPI}\_\text{TAG}\_\text{UB} \cup \{\text{MPI}\_\text{ANY}\_\text{TAG}\}, \quad 19.28 \\
\text{dtype} & : \text{MPI}\_\text{LB} .. \text{MPI}\_\text{UB} \cup \\
& \quad \{\text{MPI}\_\text{CHAR}, \text{MPI}\_\text{SHORT}, \text{MPI}\_\text{INT}, \text{MPI}\_\text{LONG}, \\
& \quad \text{MPI}\_\text{UNSIGNED}\_\text{CHAR}, \text{MPI}\_\text{UNSIGNED}\_\text{SHORT}, \\
& \quad \text{MPI}\_\text{UNSIGNED}, \text{MPI}\_\text{UNSIGNED}\_\text{LONG}, \\
& \quad \text{MPI}\_\text{FLOAT}, \text{MPI}\_\text{DOUBLE}, \text{MPI}\_\text{LONG}\_\text{DOUBLE}, \\
& \quad \text{MPI}\_\text{BYTE}, \text{MPI}\_\text{PACKED}\}, \\
\end{align*}
\]

\[
\text{nunelements} : \text{Nat}, \quad 20.12\]

\[
\text{universe} : (\text{MPI}\_\text{COMM}\_\text{WORLD} .. \text{MPI}\_\text{COMM}\_\text{WORLD} + \text{MAX}\_\text{COMM})), \quad 20.14\]

\[
\text{state} : \{\text{"send"}, \text{"recv"}\}, \quad \text{Whether this is a send or receive.} \quad 20.16\]

\[
\text{addr} : \text{Nat} \quad \text{Address in memory containing the message data.} \quad 20.18\]

\[
\text{Message}\_\text{types} \triangleq \{\text{"send"}, \text{"bsend"}, \text{"ssend"}, \text{"rsend"}, \text{"recv"}\} \quad 20.20\]

\[
\text{Collective}\_\text{types} \triangleq \{\text{"barrier"}\} \quad 20.22\]

\[
\text{Collective}\_\text{states} \triangleq \{\text{"in"}, \text{"out"}, \text{"vacant"}\} \quad 20.24\]

\[
\text{Request} \triangleq [\text{error} : \text{Nat}, \\
\quad \text{localactive} : \text{BOOLEAN}, \\
\quad \text{globalactive} : \text{BOOLEAN}, \\
\quad \text{transmitted} : \text{BOOLEAN}, \\
\quad \text{buffered} : \text{BOOLEAN}, \\
\quad \text{started} : \text{BOOLEAN}, \\
\quad \text{canceled} : \text{BOOLEAN}, \\
\quad \text{deallocated} : \text{BOOLEAN}, \\
\quad \text{ctype} : \text{Message}\_\text{types}, \\
\quad \text{persist} : \text{BOOLEAN}, \\
\quad \text{match} : \text{Seq}(\text{Nat}), \\
\quad \text{data} : \text{Nat}, \quad \text{System buffering} \\
\quad \text{message} : \text{Messages}] \quad 20.26\]

\[
\text{Requests} \triangleq [(0 .. (N - 1)) \rightarrow \text{Seq}(\text{Request})] \quad 20.28\]

\[
\text{21.45} - 21.48\]

\[\text{Status variables are explicitly allocated by the user. Therefore they are present in the Memory of individual processes. We will use a simple offset mechanism to return the individual member addresses within Memory.}\]

\[
\text{22.1} - 22.8\]

\[
\begin{align*}
\text{Status}\_\text{Canceled}(\text{base}) & \triangleq \text{base} \\
\text{Status}\_\text{Count}(\text{base}) & \triangleq \text{base} + 1 \\
\text{Status}\_\text{Source}(\text{base}) & \triangleq \text{base} + 2 \\
\text{Status}\_\text{Tag}(\text{base}) & \triangleq \text{base} + 3
\end{align*}
\]
Status_Err\( (base) \triangleq base + 4 \)

Sets the value of all status fields.

SetStatus\( (offset, canceled, count, source, tag, err) \triangleq \)

IF \( offset \% 5 = 0 \)
THEN \( canceled \)
ELSE
IF \( offset \% 5 = 1 \)
THEN \( count \)
ELSE
IF \( offset \% 5 = 2 \)
THEN \( source \)
ELSE
IF \( offset \% 5 = 3 \)
THEN \( tag \)
ELSE
IF \( offset \% 5 = 4 \)
THEN \( err \)
ELSE Assert(false, “Internal Error: Cannot have any other cases.”)

WriteMemory\( (Address, Value, Proc) \triangleq \)

\[ \text{Memory}' = [\text{Memory} \text{ EXCEPT} !\{\text{Proc} = @ \text{ EXCEPT} !\{\text{Address} = \text{Value}\}]] \]

Rank\( (\text{proc}, \text{comm}) \triangleq \) group[\text{proc}][\text{communicator}[\text{proc}][\text{comm}.\text{group}].\text{ranking}[\text{proc}]\]

Initialized \( \triangleq [0 \ldots (N - 1) \rightarrow \{“initialized”, “uninitialized”, “finalized”\}] \)

MessageBuffers \( \triangleq [0 \ldots (N - 1) \rightarrow \text{Nat}] \)

BufferSizes \( \triangleq [0 \ldots (N - 1) \rightarrow \text{Nat}] \)

Groups can be different on different processes.

Group \( \triangleq [0 \ldots (N - 1) \rightarrow 138.37 - 138.38 \)
\[ \text{MPI_COMM_WORLD} . . \]
\( (\text{MPI_COMM_WORLD} + \text{MAX_GROUP}) \rightarrow \]
\[ \{\text{members} : \text{SUBSET} (0 \ldots (N - 1)), \]
\( \text{size} : 0 \ldots N, \)
\( \text{ranking} : 0 \ldots (N - 1) \rightarrow 0 \ldots (N - 1)\}, \]
\( \text{inveranking} : 0 \ldots (N - 1) \rightarrow 0 \ldots (N - 1)\}]] \]

Communicator \( \triangleq [0 \ldots (N - 1) \rightarrow \]
\[ \text{MPI_COMM_WORLD} . . \]
\( (\text{MPI_COMM_WORLD} + \text{MAX_COMM}) \rightarrow \]
\[ \{\text{group} : \text{MPI_COMM_WORLD} . . \]
\( (\text{MPI_COMM_WORLD} + \text{MAX_GROUP}), \]
\( \text{collective} : \text{MPI_COMM_WORLD} . . \]
\( (\text{MPI_COMM_WORLD} + \text{MAX_COMM})\}]] \]
Collective $\triangleq [(\text{MPI}_\text{COMM}_\text{WORLD} \ldots
\text{MPI}_\text{COMM}_\text{WORLD} + \text{MAX}_\text{COMM})), \rightarrow
[\text{participants} : \text{subset} (0 .. (N - 1)),
\text{root} : (0 .. (N - 1)),
\text{type} : \text{Collective}\_\text{types},
\text{state} : \text{Collective}\_\text{states}]]$

$\text{Comm}\_\text{inv} \triangleq \text{communicator} \in \text{Communicator}$

$\text{Buff}\_\text{inv} \triangleq \text{bufsize} \in \text{BufferSizes}$

$\text{Msg}\_\text{buf}\_\text{inv} \triangleq \text{message}\_\text{buffer} \in \text{MessageBuffers}$

$\text{Initialized}\_\text{inv} \triangleq \text{initialized} \in \text{Initialized}$

$\text{Request}\_\text{inv} \triangleq \text{requests} \in \text{Requests}$

$\text{Col}\_\text{inv} \triangleq \text{collective} \in \text{Collective}$

$\text{group}\_\text{inv} \triangleq \text{collective} \in \text{Group}$

$\text{MPI\_Type}\_\text{Invariant} \triangleq$

$\land \text{communicator} \in \text{Communicator}$

$\land \text{bufsize} \in \text{BufferSizes}$

$\land \text{message}\_\text{buffer} \in \text{MessageBuffers}$

$\land \text{initialized} \in \text{Initialized}$

$\land \text{requests} \in \text{Requests}$

$\land \text{collective} \in \text{Collective}$

$\text{Make}\_\text{request}$ is a rule to simplify the expressions that create a new request object. Section 3.7.1

$\text{Make}\_\text{request}(\text{error}, \text{lact}, \text{gact}, \text{com}, \text{sta}, \text{buf}, \text{cty}, \text{per}, \text{mat}, \text{can}, \text{mes}) \triangleq$

$\text{error} \mapsto \text{err}$, The error code associated with this request

$\text{lact} \mapsto \text{lact}$, The message is locally awaiting completion.

$\text{gact} \mapsto \text{gact}$, The message is globally awaiting completion.

$\text{com} \mapsto \text{com}$, Data was transmitted by this message

$\text{sta} \mapsto \text{sta}$, Start this request

$\text{buf} \mapsto \text{buf}$, The data was copied from the input address

$\text{can} \mapsto \text{can}$, Whether the request was canceled

$\text{FALSE} \mapsto \text{FALSE}$, A new request is created in an allocated state

$\text{cty} \mapsto \text{cty}$, The type of message (send, bsend, rsend, or ssend)

$\text{per} \mapsto \text{per}$, Whether the request is a persistent communication

$\text{mat} \mapsto \text{mat}$, The matching process,handle

$\text{0} \mapsto \text{0}$, An uninitialized buffer

$\text{mes} \mapsto \text{mes}$, The message envelope associated with this request

The initial values for the MPI specification state variables. These are not specified by the standard, however these initial values make the TLA+ representation complete such that it can be verified using TLC.
\textit{MPI Specification Init} $\triangleq$
\begin{align*}
&\wedge \text{requests} = [i \in (0 \ldots (N - 1)) \mapsto \langle \rangle] \quad \text{Empty request sequence}, \\
&\wedge \text{bufsize} = [i \in (0 \ldots (N - 1)) \mapsto 0] \quad \text{No user attached buffer}, \\
&\wedge \text{message buffer} = [i \in (0 \ldots (N - 1)) \mapsto 0] \quad \text{No messages currently buffered}, \\
&\wedge \text{initialized} = [i \in (0 \ldots (N - 1)) \mapsto \text{“ uninitialized”}] \quad \text{Start uninitialized}, \\
&\wedge \text{communicator} = [a \in 0 \ldots (N - 1) \mapsto \\
&\quad (\text{MPI COMM WORLD} \ldots \\
&\quad (\text{MPI COMM WORLD} + \text{MAX COMM}) \mapsto \\
&\quad \{\text{group} \mapsto \text{MPI COMM WORLD}, \\
&\quad \text{collective} \mapsto i\}]] \\
&\wedge \text{collective} = [i \in (\text{MPI COMM WORLD} \ldots \\
&\quad (\text{MPI COMM WORLD} + \text{MAX COMM})) \mapsto \\
&\quad \{\text{participants} \mapsto \{\}, \\
&\quad \text{root} \mapsto 0, \\
&\quad \text{type} \mapsto \text{“ barrier"}, \\
&\quad \text{state} \mapsto \text{“ vacant”}] \\
&\wedge \forall \wedge \neg \text{RANK ORDERINGS SIGNIFICANT} \quad \text{Choose an arbitrary ordering,} \\
&\wedge \text{choose } f \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \\
&\quad \text{12.41 - 12.42 order is not specified.} \\
\end{align*}

\begin{align*}
\text{choose } finv \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \\
\forall k \in \text{domain } f : \\
&\exists n \in 0 \ldots (N - 1) : \\
&\wedge f[k] = n \\
&\wedge finv[n] = k \\
&\wedge \forall m \in \text{domain } f : f[k] = f[m] \Rightarrow k = m \\
&\wedge \text{group} = [a \in 0 \ldots (N - 1) \mapsto \\
&\quad [i \in (\text{MPI COMM WORLD} \ldots \\
&\quad ((\text{MPI COMM WORLD} + \text{MAX GROUP})) \mapsto \\
&\quad \text{IF } i = \text{MPI COMM WORLD} \\
&\quad \text{THEN} \\
&\quad \{\text{members} \mapsto \{x \in 0 \ldots (N - 1) : \text{true}\}, \\
&\quad \text{size} \mapsto N, \\
&\quad \text{ranking} \mapsto f, \\
&\quad \text{invranking} \mapsto finv\} \\
&\quad \text{ELSE} \\
&\quad \{\text{members} \mapsto \{\}, \\
&\quad \text{size} \mapsto 0, \\
&\quad \text{ranking} \mapsto [j \in 0 \ldots (N - 1) \mapsto 0], \\
&\quad \text{invranking} \mapsto [j \in 0 \ldots (N - 1) \mapsto 0]\}] \\
\vee \wedge \text{RANK ORDERINGS SIGNIFICANT} \quad \text{in this case, try all orderings,} \\
\wedge \exists f \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \\
&\exists finv \in [0 \ldots (N - 1) \mapsto 0 \ldots (N - 1)] : \\
&\quad \text{12.41 - 12.42 order is not specified.} \\
&\quad \forall k \in \text{domain } f : \\
&\quad \exists n \in 0 \ldots (N - 1) :
There is some issue with regards to where the unchanged identifiers should be living. I am using the following protocol:

1. Rules that have parameters that might be changed will declare the unchanged value appropriately inside the rule for those parameters.

2. Variables that are passed as parameters to rules must be declared as unchanged appropriately outside the rule unless the parameter might be modified by the rule when the rule is used.

3. Constants (such as a literal number, 0 for example) or constant values need not be declared as unchanged.

4. MPI based rules always indicate the unchanged terms for MPI state variables. Program models also indicate unchanged for MPI variables only when no MPI rule is fired in that transition.

Conventions on parameters.

1. Parameters that are set (i.e., OUT or INOUT) are all arrays from 0 .. (N – 1) with one instance of each object for each process in the model.

2. All other parameters (i.e., IN) are the single instance of the variable value being passed, or are constant.

3. Additional parameters are added to the end of the specified set of parameters as follows: (i) return: the address of the return variable to be written by this operation, and (ii) proc: the process that is currently executing.
These rules perform the communication or “matching” of messages that is necessary to complete the MPI communication infrastructure. They are in no way specified in the standard, except that messages are spoken of as being transmitted from one process to another and matching.

\[ \alpha \rightarrow \beta \rightarrow \text{BOOLEAN} \]

No change in state

\[ \text{Match}(a, b) \triangleq \]

\[ \land \quad \text{Assert}((a.\text{state} = \text{“recv”} \land b.\text{state} = \text{“send”}) \lor (a.\text{state} = \text{“send”} \land b.\text{state} = \text{“recv”}), \]

“Error: Match attempted with two send or receives.”

\[ \land \quad \lor a.\text{src} = b.\text{src} \quad 21.14 - 21.15 \]

\[ \lor a.\text{src} = \text{MPI\_ANY\_SOURCE} \]

\[ \lor b.\text{src} = \text{MPI\_ANY\_SOURCE} \]

\[ \land \quad a.\text{dest} = b.\text{dest} \]

\[ \land \quad a.\text{dtype} = b.\text{dtype} \quad 23.17, 23.24 - 23.27 \]

\[ \lor a.\text{msgtag} = b.\text{msgtag} \quad 21.15 - 21.16 \]

\[ \lor a.\text{msgtag} = \text{MPI\_ANY\_TAG} \]

\[ \lor b.\text{msgtag} = \text{MPI\_ANY\_TAG} \]

\[ \land \quad a.\text{universe} = b.\text{universe} \quad 19.34 - 19.37 \]

\[ \lor \quad \neg a.\text{src} = \text{MPI\_PROC\_NULL} \quad 60.48 - 61.1 \]

\[ \lor \quad \neg a.\text{dest} = \text{MPI\_PROC\_NULL} \]

\[ \lor \quad \neg b.\text{src} = \text{MPI\_PROC\_NULL} \]

\[ \lor \quad \neg b.\text{dest} = \text{MPI\_PROC\_NULL} \]

21.13 - 21.14 count need not be matched in point to point messages.

Messages match in program order pairwise between processes, however they may complete in a non-deterministic order on both the sender and receiver. This tends to imply that Communicate should in fact be two rules. And it also seems to imply that completion of a message can happen on one side and then on the other also in a non-deterministic way. Therefore Transmit should complete only one side of the communication.

Pairs messages together such that they result in a communication eventually.

\[ \text{Pair} \triangleq \]

\[ \land \exists i \in 0\.\.(N - 1) : \]

\[ \exists j \in 0\.\.(N - 1) : \]

\[ \exists m \in 1\.\.	ext{Len}(\text{requests}[i]) : \]

\[ \exists n \in 1\.\.	ext{Len}(\text{requests}[j]) : \]

\[ \text{LET} \ a \triangleq \text{requests}[i][m] \text{IN} \]

\[ \text{LET} \ b \triangleq \text{requests}[j][n] \text{IN} \]

\[ \land \ a.\text{globalactive} \]

\[ \land \ b.\text{globalactive} \]

\[ \land \ a.\text{started} \]

\[ \land \ b.\text{started} \]

\[ \land \neg a.\text{canceled} \]

\[ \land \neg b.\text{canceled} \]

\[ \land \neg a.\text{transmitted} \]
\[ \land \neg b.\text{transmitted} \]
\[ \land \lor \land a.\text{message}.\text{state} = \text{"send"} \]
\[ \land b.\text{message}.\text{state} = \text{"recv"} \]
\[ \lor \land a.\text{message}.\text{state} = \text{"recv"} \]
\[ \land b.\text{message}.\text{state} = \text{"send"} \]
\[ \land a.\text{match} = \langle \rangle \]
\[ \land b.\text{match} = \langle \rangle \]
\[ \land \text{Match}(a.\text{message}, b.\text{message}) \]
\[ \land \forall r \in 1 \ldots \text{Len}(\text{requests}[i]) : \]
\[ \land c \triangleq \text{requests}[i][r] \]
\[ \land d \triangleq \text{requests}[j][s] \]
\[ \land \lor \land c.\text{message}.\text{state} = \text{"send"} \]
\[ \land d.\text{message}.\text{state} = \text{"recv"} \]
\[ \lor \land c.\text{message}.\text{state} = \text{"recv"} \]
\[ \land d.\text{message}.\text{state} = \text{"send"} \]
\[ \land \text{Match}(c.\text{message}, d.\text{message}) \]
\[ \land c.\text{started} \]
\[ \land d.\text{started} \]
\[ \land c.\text{globalactive} \]
\[ \land d.\text{globalactive} \]
\[ \land \neg c.\text{canceled} \]
\[ \land \neg c.\text{transmitted} \]
\[ \land \neg d.\text{canceled} \]
\[ \land \neg d.\text{transmitted} \]
\[ \land c.\text{match} = \langle \rangle \]
\[ \land d.\text{match} = \langle \rangle \]
\[ \Rightarrow \land m \leq r \quad \text{Section 3.7.4} \]
\[ \land n \leq s \]
\[ \land \text{requests}' = [\text{requests} \ \text{EXCEPT} \]
\[ ![i] = \]
\[ [@ \text{EXCEPT} ![m] = \]
\[ [@ \text{EXCEPT} !.\text{match} = (j, n)], \]
\[ ![j] = \]
\[ [@ \text{EXCEPT} ![n] = \]
\[ [@ \text{EXCEPT} !.\text{match} = (i, m)]] \]
\[ \land \text{UNCHANGED} \langle \text{group}, \text{communicator}, \text{bufsize}, \text{message_buffer}, \]
\[ \text{initialized}, \text{collective} \rangle \]

Causes the communication that is already paired to complete.

Need to move arrays of data too.

\[ \text{Transmit} \triangleq \]
\[ \land \exists i \in 0 \ldots (N-1) : \]
\[ \exists j \in 1 \ldots \text{Len}(\text{requests}[i]) : \]
LET m ≡ requests[i][j]IN
∧ m.started
∧ ¬m.canceled
∧ ¬m.transmitted
∧ ∨ m.match ≠ ⟨⟩
∨ m.message.src = MPI_PROC_NULL  
∨ m.message.dest = MPI_PROC_NULL
∧ requests’ = [requests EXCEPT ![i] =
  [@ EXCEPT ![j] =
  [@ EXCEPT !.transmitted = TRUE]]
∧ LET remote ≡ requests[m.match[1]][m.match[2]]IN
  IF ∧ m.match ≠ ⟨⟩
    ∧ ¬requests[m.match[1]][m.match[2]].transmitted
  THEN
    IF m.message.state = "recv"
    THEN
      Memory’ =
      [Memory EXCEPT ![i] =
        [@ EXCEPT ![m.message.addr] =
          IF remote.buffered
            THEN remote.data Data from buffer
          ELSE Memory[m.match[1]][remote.message.addr]]
      ELSE
        Memory’ =
        [Memory EXCEPT ![m.match[1]] =
          [@ EXCEPT ![remote.message.addr] =
            IF m.buffered
              THEN m.data Data from buffer
            ELSE Memory[i][m.message.addr]]] Data from memory
    ELSE
      UNCHANGED ⟨Memory⟩
      ∧ IF m.ctype = "bsend"
    THEN
      message_buffer’ = [message_buffer EXCEPT ![i] = @ - 1]
    ELSE
      UNCHANGED ⟨message_buffer⟩
    ∧ UNCHANGED ⟨group, communicator, bufsize, initialized, collective⟩

The specification indicates that messages are buffered in an asynchronous manner. The rule Buffer is not part of the standard but necessary to allow buffering to complete asynchronously.
\[ ∧ \text{requests}[i][m].\text{started} \]
\[ ∧ \text{requests}[i][m].\text{localactive} \]
\[ ∧ \text{requests}[i][m].\text{globalactive} \]
\[ ∧ ¬\text{requests}[i][m].\text{buffered} \]
\[ ∧ ¬\text{requests}[i][m].\text{canceled} \]
\[ ∧ ¬\text{requests}[i][m].\text{transmitted} \]

Buffering is provided explicitly by the user.

\[ ∧ ∨ ∧ \text{requests}[i][m].\text{ctype} = "bsend" \]
\[ ∧ \text{requests}' = \]
\[ \text{requests} \text{ EXCEPT } ![i] = \]
\[ @ \text{ EXCEPT } ![m] = \]
\[ @ \text{ EXCEPT} \]
\[ .\text{buffered} = \text{TRUE}, \]
\[ .\text{data} = \text{Memory}[i][\text{requests}[i][m].\text{message.addr}]]] \]

Buffering may be provided by the system.

\[ ∨ ∧ \text{requests}[i][m].\text{ctype} = "send" \]
\[ ∨ ∨ \text{requests}' = \]
\[ \text{requests} \text{ EXCEPT } ![i] = \]
\[ @ \text{ EXCEPT } ![m] = \]
\[ @ \text{ EXCEPT} \]
\[ .\text{buffered} = \text{TRUE}, \]
\[ .\text{data} = \text{Memory}[i][\text{requests}[i][m].\text{message.addr}]]] \]

\[ ∨ \text{UNCHANGED} \text{ requests} \]
\[ ∧ \text{UNCHANGED} \langle \text{group, communicator, bufsize, message_buffer,} \]
\[ \text{initialized, collective} \rangle \]

General Comments:

1.19.23 – 19.24 The message source is provided in the envelope implicitly. Operators in our model must be passed this information as a parameter. As such we extend the argument list to include proc, being the unique identity of the applying process.

Section 3.2 Blocking Send and Receive Operations

Section 3.2.1 Blocking send

Can these really be done in a single transition? I am thinking that it is not possible under an interleaving semantics. In particular, either the send must be two transitions or the receive must be two transitions, it cannot be the case that they are both only one transition.

\[ \text{MPI\_Send}(\text{buf}, \text{count}, \text{datatype}, \text{dest}, \text{tag}, \text{comm}, \text{proc}) ≜ \text{MPI\_Irecv}; \text{MPI\_Wait} \]

Section 3.2.4 Blocking receive If receive is modeled using only one transition, it is just a combination of the \text{MPI\_Irecv} and Communicate rules.

\[ \text{MPI\_Recv}(\text{buf}, \text{count}, \text{datatype}, \text{source}, \text{tag}, \text{comm}, \text{status}) ≜ \text{MPI\_Irecv}; \text{MPI\_Wait} \]

Section 3.2.5 Return status

Returns in count the number of data elements in the message represented by status.
MPI Get count(status, datatype, count, return, proc) $\triangleq$ \[22.24 - 22.37\]

$\land$ Assert(Memory[proc][Status_Canceled(status)] = FALSE, "Error: count is undefined on a status from a canceled message.")

$\land$ Assert(initialized[proc] = "initialized", \[200.10 - 200.12\] "Error: MPI_Get_count called before process was initialized.")

$\land$ Memory' =
[Memory EXCEPT ![proc] =
[@ EXCEPT ![count] =
Memory[proc][Status_Count(status)]]

$\land$ UNCHANGED (group, communicator, bufsize, message_buffer, requests, initialized, collective)

Section 3.4 Communication Modes

Notes: These, like the above blocking communications really should be modeled using two transitions. In this way, the interleaving semantics is able to schedule another process to complete the communications.

Section 3.6 Buffer allocation and usage

We ignore the buffer argument as data is abstracted away in our model. Buffering is modeled as a counting semaphore, keeping track of the resources available but not exactly which resources are used or what is done with those resources.

Return value is unspecified.

MPI_Bsend(buf, count, datatype, dest, tag, comm, proc) $\triangleq$ \[< 34.17 - 34.33 >\]

$\land$ Assert(initialized[proc] = "initialized", \[200.10 - 200.12\] "Error: MPI_Bsend called with proc not in initialized state.")

$\land$ Assert(bufsize[proc] = 0, \[34.32\] "Error: MPI_Bsend called when processes buffer is non-zero.")

$\land$ buflen' = [ buflen EXCEPT ![proc] = size] \[< /34.17 - 34.33 >\]

$\land$ UNCHANGED (group, communicator, message_buffer, requests, initialized, collective)

$\land$ UNCHANGED Memory

Again we ignore the buffer_addr argument as we are abstracting data.

The standard does not indicate what the result is when there is no buffer currently attached.

MPI_Buffer_detach(buffer_addr, size, return, proc) $\triangleq$ \[< 34.36 - 35.2 >\]

$\land$ Assert(initialized[proc] = "initialized", \[200.10 - 200.12\] "Error: MPI_Buffer_detach called with proc not in initialized state.")

$\land$ Assert(bufsize[proc] $\neq$ 0,
"Error: MPI_Buffer_detach called when no associated buffer.")
∧ bufsize′ = [bufsize EXCEPT ![proc] = 0] 34.46
∧ ∀ j ∈ 1 . . . Len(requests[proc]) : 34.47
    requests[proc][j].ctype = “bsend” ⇒ requests[proc][j].transmitted
∧ Memory′ = [Memory EXCEPT ![proc] =
    [@ EXCEPT ![size] = bufsize[proc]]] 34.47
∧ UNCHANGED ⟨group, communicator, message_buffer, requests,
    initialized, collective⟩

Section 3.7.2 Communication initiation

Notes: I am not sure how to model this construct. The main problem lies in the nondeterministic
buffering scheme that the standard refers to. For a correct program one must expect no buffering,
however is it possible to write a program in such a way as to require synchronous handshakes?

Start a non-blocking standard send. 38.17 – 38.35, 58.13 – 58.18
MPI_Isend(buf, count, datatype, dest, tag, comm, request, return, proc) "send"
    “Error: MPI_Isend called with proc not in initialized state.”)
∧ Assert(proc ∈ group[proc][communicator[proc][comm].group].members,
    “Error: Proc not a member of comm group.”)
∧ LET msg "send"
    [addr → buf,
     src → Rank(proc, comm),
     dest → dest,
     msgtag → tag,
     dtype → datatype,
     numelements → count,
     universe → comm,
     state → “send”]
    IN
    requests′ = [requests EXCEPT ![proc] =
    @ o (Make_request(0, TRUE, TRUE, FALSE, TRUE, TRUE,
        “send”, FALSE, ⟨⟩, FALSE, msg))]
∧ Memory′ =
    [Memory EXCEPT ![proc] =
    [@ EXCEPT ![request] =
        Len(requests[proc]) + 1]] 40.41
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer,
    initialized, collective⟩

Set up a non-blocking buffered send. 39.1 – 39.19, 58.13 – 58.18
MPI_Ibsend(buf, count, datatype, dest, tag, comm, request, return, proc) "send"
    “Error: MPI_Ibsend called with proc not in initialized state.”)
\(\text{Assert}(\text{message\_buffer}[\text{proc}] < \text{bufsize}[\text{proc}], 28.6, 35.34 - 35.35\)

"Error: MPI\_Ibsend called when insufficient buffering was available."

\(\text{Assert}(\text{proc} \in \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm].\text{group}].\text{members},

"Error: Proc not a member of comm group."

\(\text{LET } \text{msg} \triangleq
\begin{align*}
\text{addr} & \mapsto \text{buf}, \\
\text{src} & \mapsto \text{Rank}(\text{proc}, \text{comm}), \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{"send"}
\end{align*}
\)

\(\text{LET } \text{requests}' = \left[\text{requests EXCEPT } ![\text{proc} = 40.40 \circ \langle \text{Make\_request}(0, \text{TRUE, TRUE, FALSE, TRUE, TRUE, "bsend", FALSE, \langle\rangle, \text{FALSE, msg})\rangle\right)
\)

\(\text{LET } \text{Memory}' =
\begin{align*}
\text{LET } \text{Memory EXCEPT } ![\text{proc} = \circ\]
\text{LET } \text{len}(\text{requests}[\text{proc}]) + 1\]
\end{align*}
\)

\(\text{LET } \text{message\_buffer}' = \left[\text{message\_buffer EXCEPT } ![\text{proc} = \circ + 1]\right]
\)

\(\text{SETUP a non-blocking synchronous send. 39.21 - 39.39, 58.13 - 58.18}
\)

\(\text{MPI\_Issend(\text{buf}, \text{count, datatype, dest, tag, comm, request, return, proc}) \triangleq}
\)

\(\text{LET } \text{initialized}[\text{proc}] = \text{"initialized"}, 200.10 - 200.12\)

"Error: MPI\_Issend called with proc not in initialized state."

\(\text{LET } \text{proc} \in \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}.\text{group}.\text{members},

"Error: Proc not a member of comm group."

\(\text{LET } \text{msg} \triangleq
\begin{align*}
\text{addr} & \mapsto \text{buf}, \\
\text{src} & \mapsto \text{Rank}(\text{proc}, \text{comm}), \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{"send"}
\end{align*}
\)

\(\text{LET } \text{requests}' = \left[\text{requests EXCEPT } ![\text{proc} = 40.40 \circ \langle \text{Make\_request}(0, \text{TRUE, TRUE, FALSE, TRUE, TRUE, FALSE, }
\end{align*}
\)
"ssend", false, ⟨⟩, false, msg))]

Set up a non-blocking ready send. 40.1 – 40.19, 58.13 – 58.18

MPI_Irsend(buf, count, datatype, dest, tag, comm, request, return, proc) ≜

LET msg ≜
  [addr ↦→ buf,
   src ↦→ Rank(proc, comm),
   dest ↦→ dest,
   msgtag ↦→ tag,
   dtype ↦→ datatype,
   numelements ↦→ count,
   universe ↦→ comm,
   state ↦→ "send"]

IN

LET remproc ≜ group[dest][communicator[dest][comm].group].invranking[dest]IN

  "Error: MPI_Irsend called with proc not in initialized state."

∧ Assert(proc ∈ group[proc][communicator[proc][comm].group].members,
  "Error: Proc not a member of comm group.")

∧ Assert(∃ k ∈ (1 .. Len(requests[remproc])) : 37.6 – 37.8
  ∧ requests[remproc][k].globalactive
  ∧ !requests[remproc][k].transmitted
  ∧ !requests[remproc][k].canceled
  ∧ requests[remproc][k].message.state = "recv"
  ∧ Match(msg, requests[remproc][k].message),
  "Error: MPI_Irsend started with no matching receive.")

∧ requests′ = [requests EXCEPT ![proc] = 40.40
  @ o ⟨Make_request(0, TRUE, TRUE, false, TRUE, false,
    "rsend", false, ⟨⟩, false, msg)⟩]

∧ Memory′ =
  [Memory EXCEPT ![proc] =
   [0 EXCEPT ![request] =
    Len(requests[proc]) + 1]] 40.41

∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, initialized, collective⟩

Set up a non-blocking receive. 40.21 – 40.39, 58.13 – 58.18
\[ \text{MPI\_Irecv}(\text{buf}, \text{count}, \text{datatype}, \text{source}, \text{tag}, \text{comm}, \text{request}, \text{return}, \text{proc}) \triangleq \]

\[ \wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \ 200.10 - 200.12 \]

\[ \wedge \text{"Error: MPI\_Irecv called with proc not in initialized state."} \]

\[ \wedge \text{assert}(\text{proc} \in \text{group}[\text{proc}][\text{communicator}[\text{proc}][\text{comm}].\text{group}].\text{members}, \]

\[ \text{"Error: Proc not a member of comm group."}) \]

\[ \wedge \text{LET} \ \text{msg} \triangleq \]

\[ \begin{align*}
\text{addr} & \mapsto \text{buf}, \\
\text{src} & \mapsto \text{source}, \\
\text{dest} & \mapsto \text{Rank}(\text{proc}, \text{comm}), \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{"recv"}
\end{align*} \]

\[ \text{IN} \]

\[ \text{requests'} = [\text{requests EXCEPT ![} \text{proc} = \]

\[ 0 \circ \langle \text{Make\_request}(0, \text{true}, \text{true}, \text{false}, \text{true}, \text{false}, \]

\[ \text{"recv"}, \text{false}, \langle \rangle, \text{false}, \text{msg})] \]

\[ \wedge \text{Memory'} = \]

\[ \begin{align*}
\text{[Memory EXCEPT ![} \text{proc} = \]

\[ 0 \circ \text{EXCEPT ![} \text{request} = \]

\[ \text{Len}(\text{requests}[\text{proc}]) + 1] \]

\[ \wedge \text{UNCHANGED} \langle \text{group, communicator, message\_buffer, bufsize,}

\[ \text{initialized, collective} \rangle \]

---

**Section 3.7.3 Communication Completion**

Wait for request to complete. Return information about the message in status. 41.23 – 42.6 No specification on what the status value is when a send is posted with MPI\_PROC\_NULL Specifies next state for status and request

\[ \text{MPI\_Wait}(\text{request}, \text{status}, \text{return}, \text{proc}) \triangleq \]

\[ \text{LET r} \overset{\Delta}{=} \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{request}]]\text{IN} \]

\[ \wedge \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \ 200.10 - 200.12 \]

\[ \text{"Error: MPI\_Wait called with proc not in initialized state."} \]

\[ 41.32 - 41.39 \text{ The request handle is not the null handle.} \]

\[ \wedge \wedge \text{Memory}[\text{proc}][\text{request}] \neq MPI\_REQUEST\_NULL \]

\[ \wedge r.\text{localactive} \]

\[ \wedge \forall \wedge r.\text{message}.\text{src} \neq MPI\_PROC\_NULL \]

\[ \wedge r.\text{message}.\text{dest} \neq MPI\_PROC\_NULL \]

\[ 41.32 - \text{Blocks until complete} \]

\[ \wedge \forall r.\text{transmitted} \]

\[ \forall r.\text{canceled} \]

\[ \forall r.\text{buffered} \]

\[ \wedge \text{Memory'} = \]
\[\text{Memory except } \text{proc} = 41.36 \]
\[@ \text{except } \text{Status Canceled(status)} = \]
\[r.\text{canceled} \land \neg r.\text{transmitted}, \quad 54.46 \]
\[\text{except } \text{Status Count(status)} = r.\text{message.numelements}, \]
\[\text{except } \text{Status Source(status)} = r.\text{message.src}, \]
\[\text{except } \text{Status Tag(status)} = r.\text{message.msgtag}, \]
\[\text{except } \text{Status Err(status)} = r.\text{error}, \]
\[\text{except } \text{request} = 41.32 - 41.35, 58.34 - 58.35 \]
\[\text{IF } r.\text{persist} \]
\[\text{THEN } @ \]
\[\text{ELSE } \text{MPI REQUEST NULL} \]
\[\lor \land \land r.\text{message.src} = \text{MPI PROC NULL} \]
\[\lor r.\text{message.dest} = \text{MPI PROC NULL} \]
\[\land \text{Memory}' = [\text{Memory except } \text{proc} = 41.36 \]
\[\text{except } \text{Status Canceled(status)} = r.\text{canceled}, \]
\[\text{except } \text{Status Count(status)} = 0, \]
\[\text{except } \text{Status Source(status)} = \text{MPI PROC NULL}, \]
\[\text{except } \text{Status Tag(status)} = \text{MPI ANY TAG}, \]
\[\text{except } \text{Status Err(status)} = 0, \]
\[\text{except } \text{request} = 41.32 - 41.35, 58.34 - 58.35 \]
\[\text{IF } r.\text{persist} \]
\[\text{THEN } @ \]
\[\text{ELSE } \text{MPI REQUEST NULL} \]
\[\land \text{requests}' = \]
\[\text{IF } r.\text{match} \neq \langle \rangle \]
\[\text{THEN} \]
\[\text{except } \text{requests except } \text{proc} = 58.34 \]
\[\text{except } \text{Status Canceled(status)} = r.\text{canceled}, \]
\[\text{except } \text{Status Count(status)} = 0, \]
\[\text{except } \text{Status Source(status)} = \text{MPI PROC NULL}, \]
\[\text{except } \text{Status Tag(status)} = \text{MPI ANY TAG}, \]
\[\text{except } \text{Status Err(status)} = 0, \]
\[\text{except } \text{request} = 41.32 - 41.35, 58.34 - 58.35 \]
\[\text{IF } r.\text{persist} \]
\[\text{THEN} \]
\[\text{IF } \text{requests}[r.\text{match}[1]][r.\text{match}[2]].\text{localactive} \]
\[\text{THEN } @ \text{except } !.\text{localactive} = \text{FALSE}, \]
\[!..\text{globalactive} = \text{FALSE} \]
\[\text{ELSE } @ \text{except } !.\text{localactive} = \text{FALSE} \]
\[\text{ELSE} \]
\[\text{IF } \text{requests}[r.\text{match}[1]][r.\text{match}[2]].\text{localactive} \]
\[\text{THEN } @ \text{except } !.\text{localactive} = \text{FALSE}, \]
\[!..\text{globalactive} = \text{FALSE}, \]
\[!..\text{deallocated} = \text{TRUE} \]
\[\text{ELSE } @ \text{except } !.\text{localactive} = \text{FALSE}, \]
\[!..\text{deallocated} = \text{TRUE} \]
$![$r.match[1]] =
[@ EXCEPT $![$r.match[2]] =
IF $requests[r.match[1]][r.match[2]].localactive
THEN $requests[r.match[1]][r.match[2]]
ELSE [@ EXCEPT $!.globalactive = FALSE]]

ELSE
$requests EXCEPT ![proc] = 58.34
[@ EXCEPT $![Memory[proc][request]] =
IF $r.persist
THEN [@ EXCEPT $!.localactive = FALSE]
ELSE [@ EXCEPT $!.localactive = FALSE, $!.deallocated = TRUE]]
$\land \forall Memory[proc][request] = MPI_REQUEST_NULL 41.40 - 41.41 The request handle is null or the request is not active
$\land Memory[proc][request] \neq MPI_REQUEST_NULL
$\land \neg r.localactive
$\land Memory' = [Memory EXCEPT ![proc] = 41.36
[@ EXCEPT $![StatusCanceled(status)] = FALSE,
$![StatusCount(status)] = 0,
$![StatusSource(status)] = MPI_ANY_SOURCE,
$![StatusTag(status)] = MPI_ANY_TAG,
$![StatusErr(status)] = 0]]$
$\land \text{UNCHANGED } \langle \text{requests} \rangle$
$\land \text{UNCHANGED } \langle \text{group, communicator, bufsize, message_buffer, initialized, collective} \rangle$

Test whether the request referenced has completed.
Specifies next state for request, flag, and status.

$MPI\_Test(request, flag, status, return, proc) \triangleq$

$LET r \triangleq requests[proc][Memory[proc][request]] IN$

$\land \text{Assert(initialized}[proc] = "initialized", 200.10 - 200.12$ "Error: $MPI\_Test called with proc not in initialized state." )$

$\land \forall Memory[proc][request] \neq MPI\_REQUEST\_NULL$
$\land r.localactive$ The request is active locally.
$\land \forall \land r.message.src \neq MPI\_PROC\_NULL$
$\land r.message.dest \neq MPI\_PROC\_NULL$
$42.20 - 42.21$
$\land \text{IF } \forall r.transmitted$ The communication actually happened or
$\lor r.canceled$ got canceled by the user program or
$\lor r.buffered$ got buffered.

$THEN$
$\land Memory' = [Memory EXCEPT ![proc] =
[@ EXCEPT $![StatusCanceled(status)] = \land r.canceled$
$\land \neg r.transmitted,$
\([\text{Status Count}(\text{status})] = \text{r.message.numelements},\)
\([\text{Status Source}(\text{status})] = \text{r.message.src},\)
\([\text{Status Tag}(\text{status})] = \text{r.message.msgtag},\)
\([\text{Status Err}(\text{status})] = \text{r.error},\)
\([\text{flag}] = 1,\)
\([\text{request}] =\)
\quad \text{IF } \text{r.persist} \quad \text{THEN } @ \quad \text{ELSE } \text{MPI REQUEST NULL}]

\(\wedge \text{requests}' =\)
\quad \text{IF } \text{r.match} \neq \langle \rangle \quad \text{THEN}
\quad \quad [\text{requests EXCEPT}
\quad \quad \quad ![\text{proc}] = 58.34
\quad \quad \quad [@ \text{EXCEPT } ![\text{Memory[proc][request]}] =\]
\quad \quad \quad \quad \text{IF } \text{r.persist} \quad \text{THEN}
\quad \quad \quad \quad \quad \text{IF } \text{requests[r.match[1]][r.match[2]].localactive}
\quad \quad \quad \quad \quad \quad \text{THEN } @ \text{EXCEPT } !.\text{localactive} = \text{FALSE},
\quad \quad \quad \quad \quad \quad \quad !.\text{globalactive} = \text{FALSE}
\quad \quad \quad \quad \quad \quad \text{ELSE } @ \text{EXCEPT } !.\text{localactive} = \text{FALSE}
\quad \quad \quad \quad \quad \text{ELSE}
\quad \quad \quad \quad \quad \quad \text{IF } \text{requests[r.match[1]][r.match[2]].localactive}
\quad \quad \quad \quad \quad \quad \quad @ \text{EXCEPT } !.\text{localactive} = \text{FALSE},
\quad \quad \quad \quad \quad \quad \quad !.\text{globalactive} = \text{FALSE},
\quad \quad \quad \quad \quad \quad \quad !.\text{deallocated} = \text{TRUE}
\quad \quad \quad \quad \quad \quad \text{ELSE } @ \text{EXCEPT } !.\text{localactive} = \text{FALSE},
\quad \quad \quad \quad \quad \quad \quad !.\text{deallocated} = \text{TRUE}]
\quad \quad ![\text{r.match[1]} =
\quad \quad \quad [@ \text{EXCEPT } ![\text{r.match[2]} =
\quad \quad \quad \quad \text{IF } \text{requests[r.match[1]][r.match[2]].localactive}
\quad \quad \quad \quad \quad \text{THEN } \text{requests[r.match[1]][r.match[2]]}
\quad \quad \quad \quad \quad \text{ELSE } @ \text{EXCEPT } !.\text{globalactive} = \text{FALSE}]]
\quad \quad @\text{EXCEPT } ![\text{proc}] = 58.34
\quad \quad [@ \text{EXCEPT } ![\text{Memory[proc][request]}] =\]
\quad \quad \quad \text{IF } \text{r.persist} \quad \text{THEN } @ \text{EXCEPT } !.\text{localactive} = \text{FALSE}
\quad \quad \quad \text{ELSE } @ \text{EXCEPT } !.\text{localactive} = \text{FALSE},
\quad \quad \quad \quad !.\text{deallocated} = \text{TRUE}]
\quad \quad \text{ELSE } 42.23 - 42.24 \text{ status is undefined 42.25}

\(\wedge \text{Memory}' = [\text{Memory EXCEPT ![proc]} = [@ \text{EXCEPT } ![\text{flag}] = 0]]
\wedge \text{UNCHANGED } \langle \text{requests} \rangle\)
\[ ∨ ∧ r.\text{message}.\text{src} = \text{MPI\_PROC\_NULL} \quad \text{The source or destination were} \]
\[ ∨ r.\text{message}.\text{dest} = \text{MPI\_PROC\_NULL} \quad \text{the null process 42.29 – 42.31} \]
\[ ∧ \text{Memory}' = [\text{Memory EXCEPT ![proc] =} \]
\[ @ \text{EXCEPT ![Status\_Canceled(status)]} = \text{FALSE}, \]
\[ ![\text{Status\_Count(status)}] = 0, \]
\[ ![\text{Status\_Source(status)}] = \text{MPI\_PROC\_NULL}, \]
\[ ![\text{Status\_Tag(status)}] = \text{MPI\_ANY\_TAG}, \]
\[ ![\text{Status\_Err(status)}] = 0, \]
\[ ![\text{flag}] = 1, \]
\[ ![\text{request}] = \]
\[ \text{IF } r.\text{persist} \]
\[ \text{THEN } @ \]
\[ \text{ELSE } \text{MPI\_REQUEST\_NULL]} \]
\[ ∧ \text{requests}' = \]
\[ [\text{requests EXCEPT ![proc] = 58.34} \]
\[ @ \text{EXCEPT ![Memory[proc][request]] =} \]
\[ \text{IF } r.\text{persist 42.22 – 42.23} \]
\[ \text{THEN } @ \text{EXCEPT !.localactive} = \text{FALSE}, \]
\[ !.\text{globalactive} = \text{FALSE}] \]
\[ \text{ELSE } @ \text{EXCEPT !.localactive} = \text{FALSE}, \]
\[ !.\text{globalactive} = \text{FALSE}, \]
\[ !.\text{deallocated} = \text{TRUE}]] \]
\[ ∨ ∧ \text{Memory[proc][request]} = \text{MPI\_REQUEST\_NULL} \quad \text{42.29 – 42.31} \]
\[ ∨ ∧ \text{Memory[proc][request]} \neq \text{MPI\_REQUEST\_NULL} \quad \text{The request handle is} \]
\[ ∧ ¬ r.\text{localactive} \quad \text{null or the request is not active} \]
\[ ∧ \text{Memory}' = [\text{Memory EXCEPT ![proc] =} \]
\[ @ \text{EXCEPT ![Status\_Canceled(status)]} = \text{FALSE}, \]
\[ ![\text{Status\_Count(status)}] = 0, \]
\[ ![\text{Status\_Source(status)}] = \text{MPI\_ANY\_SOURCE}, \]
\[ ![\text{Status\_Tag(status)}] = \text{MPI\_ANY\_TAG}, \]
\[ ![\text{Status\_Err(status)}] = 0, \]
\[ ![\text{flag}] = 1]] \]
\[ ∧ \text{UNCHANGED } \langle \text{requests} \rangle \]
\[ ∧ \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer,} \]
\[ \text{initialized, collective} \]

Frees the request specified.
Modifies request.
\[
\text{MPI\_Request\_free(request, return, proc) } \triangleq
\]
\[
\text{LET } \text{handle } \triangleq \text{Memory[proc][request]} \text{IN} \\
\quad ∧ \text{Assert(} \text{initialized[proc]} = \text{"initialized", 200.10 – 200.12} \]
\quad "\text{MPI\_Request\_free called with proc not in initialized state."} \)
\[
\quad ∧ \text{Assert(requests[proc][handle].ctype } \neq \text{"recv", 43.37 – 43.39} \]
\quad "\text{MPI\_Request\_free called with a receive request."} \)
Assert( \forall \text{requests}[\text{proc}][\text{handle}].\text{message}.\text{dest} = \text{MPI_PROC_NULL} \land \neg \text{requests}[\text{proc}][\text{handle}].\text{match} \neq \emptyset, \land \neg \text{MPI_Request_free called with an unmatched request.} )

\land \text{assertions } (61.1

\land \text{requests}[\text{proc}][\text{handle}].\text{message}.\dest = \text{MPI_PROC_NULL} \land \exists \text{i} \in (0 \ldots (N - 1)) :

\land \text{r(i).localactive} \land \text{r(i).match[1]} \land \text{r(i).match[2]}

\land \neg \text{requests[i][j].localactive},

\text{"Error: incomplete request freed."}

\land \text{Memory'} = \text{Memory EXCEPT ![\text{proc}] = [@ EXCEPT ![\text{request}] = \text{MPI_REQUEST_NULL}]}

\land \\text{requests'} = \text{requests EXCEPT ![\text{proc}] = [@ EXCEPT ![\text{Memory}[\text{proc}][\text{request}]] = [\neg \text{localactive} = \text{false}, [@ \text{globalactive} = \text{false}]]]

\land \text{UNCHANGED } \langle \text{group, communicator, bufsize, message_buffer, initialized, collective} \rangle

Section 3.7.5 Multiple Completions

\land \forall \exists \text{i} \in (0 \ldots (\text{count} - 1)) :

\land \text{Memory}[\text{proc}][\text{array_of_requests} + \text{i}] \neq \text{MPI_REQUEST_NULL}

\land \text{r(i).localactive} \land \text{r(i).message.src} \neq \text{MPI_PROC_NULL} \land \text{r(i).message.src} \neq \text{MPI_PROC_NULL} \land \forall \text{r(i).transmitted} \land \forall \text{r(i).canceled} \land \forall \text{r(i).buffered}

\land \text{Memory'} = [\text{Memory EXCEPT ![\text{proc}]} = \text{45.46} - 45.47

\land [\text{Status.Source(status)} = \text{r(i).message.src}

\land [\text{Status.Tag(status)} = \text{r(i).message.msgtag},

\text{Wait for one of the requests referenced in array_of_requests to complete.}

\text{Specifies next state for index and status }

\text{MPI.Waitany(count, array_of_requests, index, status, return, proc) } \triangleq

\text{LET } \text{r(v) } \triangleq \text{requests[proc][Memory[proc][array_of_requests + v]]IN

\land \text{assertions } (200.12

\text{Error: MPI.Waitany called with proc not in initialized state."})

\text{45.44 \text{Blocks } \text{chooses nondeterministically.}

\land \forall \exists \text{i} \in (0 \ldots (\text{count} - 1)) :

\land \text{Memory[proc][array_of_requests} + \text{i}] \neq \text{MPI_REQUEST_NULL}

\land \text{r(i).localactive} \land \text{r(i).message.src} \neq \text{MPI_PROC_NULL} \land \text{r(i).message.src} \neq \text{MPI_PROC_NULL} \land \forall \text{r(i).transmitted} \land \forall \text{r(i).canceled} \land \forall \text{r(i).buffered}

\land \text{Memory'} = [\text{Memory EXCEPT ![\text{proc}]} = \text{45.46} - 45.47

\land [\text{Status.Source(status)} = \text{r(i).message.src}

\land [\text{Status.Tag(status)} = \text{r(i).message.msgtag},

\text{43.20 Not modeling deallocation

\text{1069 \land \neg \text{MPI.Request_free called with an unmatched request."})

\text{43.35 Not modeling deallocation

\text{1069 \land \neg \text{MPI.Request_free called with an unmatched request."})}
\text{Status \ Err}(\text{status}) = r(i).error, \\
\text{Status \ Count}(\text{status}) = r(i).message.numelements, \\
\text{Status \ Canceled}(\text{status}) = \land r(i).canceled, \\
\land \neg r(i).transmitted, \\
\text{array \ of \ requests} + i = 46.1 - 46.2, 58.34 - 58.35 \\
\text{IF } r(i).\text{persist} \\
\quad \text{THEN } @ \\
\quad \text{ELSE } \text{MPI\_REQUEST\_NULL}, \\
\quad ![\text{index} = i] 45.46 \\
\lor \land \lor r(i).message.src = \text{MPI\_PROC\_NULL} \quad \text{The src or dest was} \\
\lor r(i).message.dest = \text{MPI\_PROC\_NULL} \quad \text{the null process} \\
\land \text{Memory}' = [\text{Memory \ EXCEPT} ![\text{proc}] = [@ \text{EXCEPT} \\
\quad ![\text{Status \ Source}(\text{status})] = \text{MPI\_PROC\_NULL}, \\
\quad ![\text{Status \ Tag}(\text{status})] = \text{MPI\_ANY\_TAG}, \\
\quad ![\text{Status \ Err}(\text{status})] = 0, \\
\quad ![\text{Status \ Count}(\text{status})] = 0, \\
\quad ![\text{Status \ Canceled}(\text{status})] = r(i).canceled, \\
\quad ![\text{array \ of \ requests} + i] = 46.2, 58.34 - 58.35 \\
\text{IF } r(i).\text{persist} \\
\quad \text{THEN } @ \\
\quad \text{ELSE } \text{MPI\_REQUEST\_NULL}, \\
\quad ![\text{index} = i] 45.46 \\
\land \text{requests}' = \\
\text{IF } r(i).\text{match} \neq \langle \rangle 46.30 - 46.31, 58.34 \\
\text{THEN} \\
\quad ![\text{requests} \text{ EXCEPT} \\
\quad ![\text{proc}] = \\
\quad [@ \text{EXCEPT} ![\text{Memory}[\text{proc}][\text{array \ of \ requests} + i]] = \\
\quad \text{IF } r(i).\text{persist} \\
\quad \text{THEN} \\
\quad \quad \text{IF } \text{requests}[r(i).\text{match}[1]]r(i).\text{match}[2].\text{localactive} \\
\quad \quad \text{THEN } [@ \text{EXCEPT} !.\text{localactive} = \text{FALSE}, \\
\quad \quad \quad !.\text{globalactive} = \text{FALSE}] \\
\quad \quad \text{ELSE } [@ \text{EXCEPT} !.\text{localactive} = \text{FALSE}] \\
\quad \quad \text{ELSE} \\
\quad \quad \text{IF } \text{requests}[r(i).\text{match}[1]]r(i).\text{match}[2].\text{localactive} \\
\quad \quad \text{THEN } [@ \text{EXCEPT} !.\text{localactive} = \text{FALSE}, \\
\quad \quad \quad !.\text{globalactive} = \text{FALSE}, \\
\quad \quad \quad !.\text{deallocated} = \text{TRUE}] \\
\quad \quad \text{ELSE } [@ \text{EXCEPT} !.\text{localactive} = \text{FALSE}, \\
\quad \quad \quad !.\text{deallocated} = \text{TRUE}], \\
\quad ![r(i).\text{match}[1]] = \\
\quad [@ \text{EXCEPT} ![r(i).\text{match}[2]] =
\[
\begin{align*}
\text{IF } & \text{ requests}[r(i).match[1]][r(i).match[2]].\text{localactive} \\
& \text{ THEN } \text{ requests}[r(i).match[1]][r(i).match[2]] \\
& \text{ ELSE } [@ \text{ EXCEPT ![proc] = FALSE}] \\
\text{ELSE } & \text{ requests EXCEPT ![proc] = } \\
& [@ \text{ EXCEPT ![Memory[proc][array_of_requests + i]] = } \\
& \text{ IF } r(i).\text{persist} \\
& \text{ THEN } [@ \text{ EXCEPT ![proc] = FALSE}] \\
& \text{ ELSE } [@ \text{ EXCEPT ![proc] = FALSE,} \\
& \text{ ![deallocated] = TRUE] } \\
\forall \forall i \in (0..(count - 1)) : \\
\land \land Memory[proc][array_of_requests + i] = \text{MPI_REQUEST_NULL} \\
\lor \land Memory[proc][array_of_requests + i] \neq \text{MPI_REQUEST_NULL} \\
\land \neg r(i).localactive \\
\land Memory' = [\text{Memory EXCEPT ![proc] = } \\
[@ \text{ EXCEPT } \\
\text{ ![Status\_Source(status)] = \text{MPI\_ANY\_SOURCE},} \\
\text{ ![Status\_Tag(status)] = \text{MPI\_ANY\_TAG},} \\
\text{ ![Status\_Err(status)] = 0,} \\
\text{ ![Status\_Count(status)] = 0,} \\
\text{ ![Status\_Canceled(status)] = FALSE,} \\
\text{ ![index] = \text{MPI\_UNDEFINED}] } \\
\land \text{UNCHANGED } \langle \text{requests} \rangle \\
\land \text{UNCHANGED } \langle \text{group, communicator, bufsize, message_buffer,} \\
\text{ initialized, collective} \rangle \\
\text{Test whether one of the requests referenced in array_of_requests has completed.} \\
\text{MPI\_Testany(count, array_of_requests, index, flag, status, return, proc) } \triangleq \\
\text{LET } r(v) \triangleq \text{requests[proc][Memory[proc][array_of_requests + v]IN} \\
\text{ Assert(initialized[proc] = "initialized", 200.10 - 200.12} \\
\text{ Error: MPI\_Testany called with proc not in initialized state."}) \\
\land \lor \forall i \in (0..(count - 1)) : \\
\land Memory[proc][array_of_requests + i] \neq \text{MPI\_REQUEST\_NULL} \\
\land r(i).localactive \\
\land \lor \land r(i).message.src \neq \text{MPI\_PROC\_NULL} \text{ The source is not null} \\
\land r(i).message.dest \neq \text{MPI\_PROC\_NULL} \text{ The destination is not null} \\
\land \text{IF } \lor r(i).transmitted \text{ The communication happened or} \\
\lor r(i).canceled \text{ got canceled or} \\
\lor r(i).buffered \text{ got buffered.} \\
\text{ THEN } \\
\land Memory' = [\text{Memory EXCEPT ![proc] = } [@ \text{ EXCEPT } \\
\text{ ![flag] = 1, 46.29 using C convention for boolean} \\
\text{ ![Status\_Source(status)] = r(i).message.src, 46.30} \\
\text{ ![Status\_Tag(status)] = r(i).message.msgtag,} \\
\text{ ![Status\_Err(status)] = r(i).error,}
\end{align*}
\]
\[\begin{align*}
\text{Status \_ Count}(status) &= r(i).message.numelements, \\
\text{Status \_ Canceled}(status) &= \land r(i).canceled \land \neg r(i).transmitted, \\
\text{index} &= i, \\
\text{array \_ of \_ requests} + i &= 46.31 - 46.32, 58.34 - 58.35 \\
\text{if } r(i).persist \text{ THEN } @ \\
\text{else } \text{MPI\_REQUEST\_NULL}] \\
\land requests' = \\
\text{let } remreq \triangleq requests[r(i).match[1]]r(i).match[2]\text{IN} \\
\text{if } r(i).match \neq \langle \rangle \text{ THEN} \\
\text{requests EXCEPT ![proc] =} \\
[@ \text{EXCEPT } ![\text{Memory}[proc][\text{array \_ of \_ requests} + i]] = \\
\text{IF } r(i).persist \text{ THEN} \\
\text{IF } remreq.localactive \text{ THEN} \\
[@ \text{EXCEPT } !.localactive = FALSE, \\
! .globalactive = FALSE] \\
\text{ELSE} \\
[@ \text{EXCEPT } !.localactive = FALSE] \\
\text{ELSE} \\
\text{IF } remreq.localactive \text{ THEN} \\
[@ \text{EXCEPT } !.localactive = FALSE, \\
! .globalactive = FALSE, \\
! .deallocated = TRUE] \\
\text{ELSE} \\
[@ \text{EXCEPT } !.localactive = FALSE, \\
! .deallocated = TRUE]], \\
! [r(i).match[1]] = \\
[@ \text{EXCEPT } ![r(i).match[2]] = \\
\text{IF } remreq.localactive \text{ THEN} \\
\text{requests}[r(i).match[1]]r(i).match[2]] \\
\text{ELSE} \\
[@ \text{EXCEPT } !.globalactive = FALSE]] \\
\text{ELSE} \\
\text{requests EXCEPT ![proc] =} \\
[@ \text{EXCEPT } ![\text{Memory}[proc][\text{array \_ of \_ requests} + i]] = \\
\text{IF } r(i).persist \text{ THEN} 
\end{align*}\]
\[\@ \text{EXCEPT} .\text{localactive} = \text{false}\]

ELSE

\[\@ \text{EXCEPT} .\text{localactive} = \text{false},
.\text{deallocated} = \text{true}]\]

ELSE

\[\wedge \text{Memory}' =
\[\@ \text{EXCEPT} ![\text{proc}] =
\[\@ \text{EXCEPT} \text{status is explicitly undefined},
![\text{flag}] = 0, 46.33
![\text{index}] = \text{MPI\_UNDEFINED}] 46.33 - 46.34
\wedge \text{UNCHANGED } \langle \text{requests} \rangle\]

\[\vee \wedge \text{r}(i).\text{message.src} = \text{MPI\_PROC\_NULL} \quad \text{The source or dest were}\]
\[\vee \text{r}(i).\text{message.dest} = \text{MPI\_PROC\_NULL} \quad \text{the null process 61.3 - 61.4}\]
\[\wedge \text{Memory}' = \lbrack \text{Memory EXCEPT ![\text{proc}] = } \lbrack @ \text{EXCEPT}
![\text{flag}] = 1, 46.29
![\text{Status\_Source(status)}] = \text{MPI\_PROC\_NULL},
![\text{Status\_Tag(status)}] = \text{MPI\_ANY\_TAG},
![\text{Status\_Err(status)}] = 0,
![\text{Status\_Canceled(status)}] = \text{r}(i).\text{canceled},
![\text{index}] = i, 46.29
![\text{array_of_requests + i}] =
\text{IF } \text{r}(i).\text{persist}
\text{THEN } @
\text{ELSE } \text{MPI\_REQUEST\_NULL}] 46.31 - 46.32, 58.34 - 58.35
\wedge \text{requests}' = 46.31 - 46.32, 58.34 - 58.35
\[\@ \text{EXCEPT} ![\text{proc}] =
\lbrack @ \text{EXCEPT} ![\text{Memory[\text{proc}][\text{array_of_requests + i}]} =
\lbrack @ \text{EXCEPT} !.\text{localactive} = \text{false}]\]
\[\forall i \in (0 \ldots (\text{count} - 1)):
46.35 - 46.37
\wedge \text{Memory[\text{proc}][\text{array_of_requests + i}] = MPI\_REQUEST\_NULL}
\wedge \text{Memory[\text{proc}][\text{array_of_requests + i}] \neq MPI\_REQUEST\_NULL}
\wedge \neg \text{r}(i).\text{localactive} \quad \text{request is not active}\]
\[\wedge \text{Memory}' = \lbrack \text{Memory EXCEPT ![\text{proc}] = } \lbrack @ \text{EXCEPT}
![\text{flag}] = 1, 46.36
![\text{Status\_Source(status)}] = \text{MPI\_ANY\_SOURCE}, 46.36
![\text{Status\_Tag(status)}] = \text{MPI\_ANY\_TAG},
![\text{Status\_Err(status)}] = 0,
![\text{Status\_Canceled(status)}] = \text{false},
![\text{index}] = \text{MPI\_UNDEFINED}] 46.36
\wedge \text{UNCHANGED } \langle \text{requests} \rangle\]
\[\wedge \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer,} \]
A long version of \textit{MPI\_Waitall} – includes the line by line reference. Specifies the next state for \texttt{array\_of\_requests} and \texttt{array\_of\_statuses}.

\begin{verbatim}

MPI\_Waitall\((\text{count}, \text{array\_of\_requests}, \text{array\_of\_statuses}, \text{return}, \text{proc})\) \triangleq

\text{LET } r(v) \triangleq \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{array\_of\_requests} + v]]\text{IN}

\text{\& Assert (initialized[proc] = "initialized", 200.10 – 200.12)
}

\text{\& \forall i \in (0 .. (\text{count} - 1)) : 47.18

Empties status for \texttt{requests} = MPI\_REQUEST\_NULL

\& \text{\& Memo[\text{proc}][\text{array\_of\_requests} + i] = MPI\_REQUEST\_NULL

\& \text{\& Memo[\text{proc}][\text{array\_of\_requests} + i] \neq MPI\_REQUEST\_NULL

\& \text{\& \neg r(i).localactive Empties status when not locally active

\& \text{\& Memo[\text{proc}][\text{array\_of\_requests} + i] \neq MPI\_REQUEST\_NULL

\& \text{\& r(i).message.src \neq MPI\_PROC\_NULL The source is not null

\& \text{\& r(i).message.dest \neq MPI\_PROC\_NULL The destination is not null

\& \text{\& r(i).transmitted The communication actually happened or

\& \text{\& r(i).canceled the communication got canceled or

\& \text{\& r(i).buffered the communication got buffered.

\& \text{\& Memo[\text{proc}][\text{array\_of\_requests} + i] \neq MPI\_REQUEST\_NULL

\& \text{\& \& r(i).message.src = MPI\_PROC\_NULL The source or destination was

\& \text{\& \& r(i).message.dest = MPI\_PROC\_NULL the null process

\text{\& Memo'[} = 47.18 - 47.21

\text{\& Memo EXCEPT ![\text{proc}]} = 47.22 - 47.23

\text{[j \in 1 .. \text{Len(Memo[\text{proc}])} \mapsto

\text{IF } j \in \text{array\_of\_requests} .. (\text{array\_of\_requests} + (\text{count} - 1))

\text{THEN

\text{IF } \text{\& Memo[\text{proc}][j] = MPI\_REQUEST\_NULL

\text{\& Memo[\text{proc}][j] \neq MPI\_REQUEST\_NULL

\text{\& \neg requests[\text{proc}][\text{Memo[\text{proc}][j]].localactive

\text{THEN Memo[\text{proc}][j]

\text{ELSE

\text{IF requests[\text{proc}][\text{Memo[\text{proc}][j]].persist

\text{THEN @

\text{ELSE MPI\_REQUEST\_NULL

\text{ELSE

\text{LET offset \triangleq j - array\_of\_statuses\text{IN

\text{LET reqj \triangleq requests

\text{\[\text{proc}

\text{\[\text{Memo[\text{proc}][\text{array\_of\_requests} + ((\text{offset} \div 5))\text{IN

\text{IF } j \in \text{array\_of\_statuses} .. (\text{array\_of\_statuses} + ((\text{count} \times 5) - 1))

\text{THEN

\text{IF } \text{\& Memo

\text{[\text{proc]}

\end{verbatim}

\text{}}}
\([\text{array} \text{ of } \text{requests} + (\text{offset} \div 5)] = \text{MPI\_REQUEST\_NULL}\)
\(\lor \neg \text{req}j.\text{localactive}\)

THEN
\(\text{SetStatus}(\text{offset},\)
\(\text{FALSE},\)
\(0,\)
\(\text{MPI\_ANY\_SOURCE},\)
\(\text{MPI\_ANY\_TAG},\)
\(0)\)

ELSE
IF \(\lor \text{req}j.\text{message}.\text{src} = \text{MPI\_PROC\_NULL}\)
\(\lor \text{req}j.\text{message}.\text{dest} = \text{MPI\_PROC\_NULL}\)
THEN \(61.3 - 61.4\)
\(\text{SetStatus}(\text{offset},\)
\(\text{req}j.\text{canceled} \land \neg \text{req}j.\text{transmitted},\)
\(0, \text{MPI\_PROC\_NULL}, \text{MPI\_ANY\_TAG}, 0)\)
ELSE \(\text{SetStatus}(\text{offset},\)
\(\text{req}j.\text{canceled} \land \neg \text{req}j.\text{transmitted},\)
\(\text{req}j.\text{message}.\text{numelements},\)
\(\text{req}j.\text{message}.\text{src},\)
\(\text{req}j.\text{message}.\text{msgtag},\)
\(\text{req}j.\text{error})\)
ELSE \(\text{Memory}[\text{proc}][j]\)
\(\land \text{requests}' =\)
\([j \in 0 .. (N - 1) \mapsto 47.22, 58.34 \text{ Not modeling deallocation}\)
IF \(j = \text{proc}\)
THEN
IF \(\exists m \in 0 .. (\text{count} - 1) :\)
\(\land \text{Memory}[\text{proc}][\text{array} \text{ of } \text{requests} + m] \neq \text{MPI\_REQUEST\_NULL}\)
\(\land k = \text{Memory}[\text{proc}][\text{array} \text{ of } \text{requests} + m]\)
THEN
IF \(\text{requests}[j][k].\text{match} \neq \langle \rangle\)
THEN
IF \(\text{requests}[\text{requests}[j][k].\text{match}[1]]\)
\(\text{requests}[j][k].\text{match}[2] \land \text{localactive}\)
THEN
\([\text{requests}[j][k] \text{ EXCEPT } \neg \text{localactive} = \text{FALSE},\)
\(\neg \text{globalactive} = \text{FALSE}]\)
ELSE
\([\text{requests}[j][k] \text{ EXCEPT } \neg \text{localactive} = \text{FALSE}]\)
ELSE
\([\text{requests}[j][k] \text{ EXCEPT } \neg \text{localactive} = \text{FALSE}]\)
ELSE 

\[ k \in 1 \ldots \text{Len}(\text{requests}[j]) \mapsto \]

IF \( \exists m \in 0 \ldots (\text{count} - 1) : \)

\( \land \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + m] \neq \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \land \langle j, k \rangle = r(m).\text{match} \)

THEN

IF \( \neg r(i).\text{localactive} \)

THEN

\[ [\text{requests}[j][k] \text{EXCEPT} !.\text{globalactive} = \text{FALSE}] \]

ELSE

\( \text{requests}[j][k] \)

ELSE

\( \text{requests}[j][k]] \)

\( \land \text{UNCHANGED} \langle \text{group}, \text{communicator}, \text{bufsize}, \text{message} \_\text{buffer}, \)

\( \text{initialized}, \text{collective} \rangle \)

Test whether all requests referenced in \( \text{array} \_\text{of} \_\text{requests} \) have completed.

\( \text{MPI} \_\text{Testall}(\text{count}, \text{array} \_\text{of} \_\text{requests}, \text{flag}, \text{array} \_\text{of} \_\text{statuses}, \text{return}, \text{proc}) \triangleq \)

LET \( r(v) \triangleq \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + v] \text{IN} \)

\( \land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \)

\( \text{"Error: MPI} \_\text{Testall called with proc not in initialized state."} \)\)

\( \land \text{IF} \ \forall i \in (0 \ldots (\text{count} - 1)) : \)

\( \text{Empty} \text{status for} \text{requests} = \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \land \lor \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + i] = \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \lor \land \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + i] \neq \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \land \neg r(i).\text{localactive} \) \( \text{Empty} \text{status when not} \text{locally} \text{active} \)

\( \lor \land \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + i] \neq \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \land r(i).\text{message} \_\text{src} \neq \text{MPI} \_\text{PROC} \_\text{NULL} \) \( \text{The} \text{source is not} \text{null} \)

\( \land r(i).\text{message} \_\text{dest} \neq \text{MPI} \_\text{PROC} \_\text{NULL} \) \( \text{The} \text{destination is not} \text{null} \)

\( \land \lor r(i).\text{transmitted} \) \( \text{The} \text{communication} \text{actually} \text{happened} \text{or} \)

\( \lor r(i).\text{canceled} \) \( \text{the} \text{communication} \text{got} \text{canceled} \text{or} \)

\( \lor r(i).\text{buffered} \) \( \text{the} \text{communication} \text{got} \text{buffered}. \)

\( \lor \land \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + i] \neq \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \land \lor r(i).\text{message} \_\text{src} = \text{MPI} \_\text{PROC} \_\text{NULL} \) \( \text{The} \text{src} \text{or} \text{dest} \text{was} \)

\( \lor r(i).\text{message} \_\text{dest} = \text{MPI} \_\text{PROC} \_\text{NULL} \) \( \text{the} \text{null} \text{process} \)

\( \forall i \in (0 \ldots (\text{count} - 1)) : \) \( 48.16 \)

\( \land \lor \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + i] = \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \lor \land \text{Memory}[\text{proc}][\text{array} \_\text{of} \_\text{requests} + i] \neq \text{MPI} \_\text{REQUEST} \_\text{NULL} \)

\( \land \neg r(i).\text{localactive} \)

THEN

\( \land \text{Memory}' = \ 47.18 - 47.21 \)

\( [\text{Memory} \ \text{EXCEPT} \ ![\text{proc}] = 47.22 - 47.23 \)
\[ j \in 1 \ldots \text{Len}(\text{Memory}[\text{proc}]) \mapsto \]

\[
\text{IF } j \in \text{array_of_requests} \ldots (\text{array_of_requests} + (\text{count} - 1)) \text{ THEN}
\]

\[
\text{IF } \lor \text{Memory}[\text{proc}][j] = \text{MPI_REQUEST_NULL} \land \text{Memory}[\text{proc}][j] \neq \text{MPI_REQUEST_NULL} \land \neg \text{requests}[\text{proc}][\text{Memory}[\text{proc}][j]].\text{localactive}
\]

\[
\text{THEN Memory}[\text{proc}][j]
\]

\[
\text{ELSE}
\]

\[
\text{IF requests}[\text{proc}][\text{Memory}[\text{proc}][j]].\text{persist} \text{ THEN } @\text{ ELSE } \text{MPI_REQUEST_NULL}
\]

\[
\text{ELSE}
\]

\[
\text{LET offset} = j - \text{array_of_statuses}\text{IN}
\]

\[
\text{LET reqj} = \text{requests}[\text{proc}][\text{array_of_requests} + ((\text{offset}) \div 5)]\text{IN}
\]

\[
\text{IF j} \in \text{array_of_statuses} \ldots (\text{array_of_statuses} + ((\text{count} \times 5) - 1)) \text{ THEN}
\]

\[
\text{IF } \lor \text{Memory}[\text{proc}][\text{array_of_requests} + (\text{offset} \div 5)] = \text{MPI_REQUEST_NULL} \land \neg \text{reqj}.\text{localactive}
\]

\[
\text{THEN SetStatus}(\text{offset},
\]

\[
\text{FALSE, 0,}
\]

\[
\text{MPI_ANY_SOURCE,}
\]

\[
\text{MPI_ANY_TAG, 0)}\]

\[
\text{ELSE}
\]

\[
\text{IF } \lor \text{reqj}.\text{message}.\text{src} = \text{MPI_PROC_NULL} \land \text{reqj}.\text{message}.\text{dest} = \text{MPI_PROC_NULL}
\]

\[
\text{THEN } 61.3 - 61.4 \text{ SetStatus}(\text{offset},
\]

\[
\text{FALSE, 0,}
\]

\[
\text{MPI_PROC_NULL,}
\]

\[
\text{MPI_ANY_TAG, 0)}\]

\[
\text{ELSE SetStatus}(\text{offset}, 48.17 - 48.18
\]

\[
\text{reqj}.\text{canceled} \land \neg \text{reqj}.\text{transmitted,}
\]

\[
\text{reqj}.\text{message}.\text{nuelements,}
\]

\[
\text{reqj}.\text{message}.\text{src,}
\]

\[
\text{reqj}.\text{message}.\text{msgtag,}
\]
ELSE
    IF \( j = \text{flag} \)
    THEN 48.15
    ELSE \( \text{Memory}[\text{proc}][j] \)\)
\( \land requests' = 48.18 - 48.19, 58.34 \) Not modeling deallocation
\( [j \in 0 \ldots (N - 1) \mapsto 47.22, 58.34 \) Not modeling deallocation
\( \) IF \( j = \text{proc} \)
THEN
\( [k \in 1 \ldots \text{Len}(requests[j]) \mapsto \)
  IF \( \exists m \in 0 \ldots (\text{count} - 1) : \)
  \( \land \text{Memory} \)
  \( [\text{proc}][\text{array_of_requests} + m] \neq \text{MPI_REQUEST_NULL} \)
  \( \land k = \text{Memory}[\text{proc}][\text{array_of_requests} + m] \)
THEN
  IF \( \text{requests}[j][k].\text{match} \neq \{\} \)
  THEN
    IF \( \text{requests}[\text{requests}[j][k].\text{match}[1]] \)
    THEN
      \( [\text{requests}[j][k].\text{match}[2]].\text{localactive} \)
    THEN
      \( [\text{requests}[j][k] \text{ EXCEPTION } !.\text{localactive} = \text{FALSE}, \)
      \( !.\text{globalactive} = \text{FALSE}] \)
    ELSE
      \( [\text{requests}[j][k] \text{ EXCEPTION } !.\text{localactive} = \text{FALSE}] \)
    ELSE
      \( [\text{requests}[j][k] \text{ EXCEPTION } !.\text{localactive} = \text{FALSE}] \)
    ELSE
      \( \text{requests}[j][k] \)
  ELSE
  \( [k \in 1 \ldots \text{Len}(requests[j]) \mapsto \)
  IF \( \exists m \in 0 \ldots (\text{count} - 1) : \)
  \( \land \text{Memory} \)
  \( [\text{proc}][\text{array_of_requests} + m] \neq \text{MPI_REQUEST_NULL} \)
  \( \land \langle j, k \rangle = r(m).\text{match} \)
THEN
  IF \( \neg \text{requests}[j][k].\text{localactive} \)
  THEN
    \( [\text{requests}[j][k] \text{ EXCEPTION } !.\text{globalactive} = \text{FALSE}] \)
  ELSE
  \( \text{requests}[j][k] \)
ELSE

requests[j][k]]

ELSE

∧ Memory′ = [Memory EXCEPT ![proc] =

[@ EXCEPT ![flag] = 0]] 48.21 – 48.22

∧ UNCHANGED (requests)

∧ UNCHANGED ⟨group, communicator, bufsise, message_buffer, initialized, collective⟩

Wait for some subset of the requests referenced in array_of_requests to complete. The ordering of array_of_indices or array_of_statuses is not specified. Not modeling the possibility of arbitrary ordering of the array_of_indices or array_of_statuses.

MPI_Waitsome(incount, array_of_requests, outcount,

array_of_indices, array_of_statuses, return, proc) ≜

LET r(v) ≜ requests[proc][Memory[proc][array_of_requests + v]] [IN

LET msgs ≜ The set of messages that

{x ∈ (0 . . (incount − 1)) : have completed in array_of_requests

∧ Memory The request

[proc] handle is not the null handle

[array_of_requests + x] ≠ MPI_REQUEST_NULL

∧ r(x).localactive The request is active.

∧ ∀ r(x).transmitted The communication actually happened or

∀ r(x).canceled the communication got canceled or

∀ r(x).buffered} the communication got buffered

IN


"Error: MPI_Waitsome called with proc not in initialized state.")

∧ ∀ ∀ Cardinality(msgs) > 0 48.45

∧ ∃ seq ∈ [1 . . Cardinality(msgs) → msgs] :

∧ ∀ s ∈ msgs :

∃ n ∈ 1 . . Len(seq) :

∧ seq[n] = s

∧ ∀ m ∈ 1 . . Len(seq) : seq[n] = seq[m] ⇒ m = n

∧ Memory′ =

[Memory EXCEPT ![proc] =

[j ∈ 1 . . Len(Memory[proc])] →

IF j ∈ array_of_requests . . (array_of_requests + (incount − 1))

THEN

IF ∧ ∃ m ∈ msgs : ∧ j = (array_of_requests + m)

∧ ¬r(m).persist

THEN MPI_REQUEST_NULL

ELSE Memory[proc][j]

ELSE

IF ∧ j ∈ array_of_indices . . (array_of_indices + (incount − 1))

∧ (j − array_of_indices) < Len(seq)
then
seq[j - array_of_indices + 1]
ELSE
IF \( \land j \in array_of_statuses \) . .
\( (array_of_statuses + ((\text{incout} \ast 5) - 1)) \)
\( \land ((j - array_of_statuses) \div 5) < \text{Len}(seq) \)
THEN
LET offset \( \triangleq j - array_of_statuses \) IN
LET reqj \( \triangleq r(seq[((\text{offset}) \div 5) + 1])) \) IN
IF \( \lor \text{reqj.message.src} = \text{MPI_PROC_NULL} \)
\( \lor \text{reqj.message.dest} = \text{MPI_PROC_NULL} \)
THEN SetStatus(offset, 61.3 – 61.4)
    FALSE,
    0,
    MPI_PROC_NULL,
    MPI_ANY_TAG,
    0)
ELSE SetStatus(offset, 48.17 – 48.18)
    reqj.canceled \( \land \neg\text{reqj.transmitted}, \)
    reqj.message.numelements,
    reqj.message.src,
    reqj.message.msgtag,
    reqj.error)
ELSE
IF j = outcount
THEN Cardinality(msgs) 48.46
ELSE Memory[proc][j][]
\( \land requests' = \)
[\( i \in (0 . . (N - 1)) \) \( \mapsto \)
IF i = proc
THEN
\[ j \in 1 . . \text{Len}(requests[i]) \mapsto \]
IF \( \exists m \in \text{msgs} : \)
\( \land j = \text{Memory}[i][\text{array_of_requests} + m] \)
THEN
IF requests[i][j].match \( \neq \langle \rangle \)
THEN
IF requests[requests[i][j].match[1]]
\[ requests[i][j].match[2]],\text{localactive} \]
THEN
\[ requests[i][j] \) EXCEPT \!).localactive = \text{FALSE},
\)\!).globalactive = \text{FALSE}]
ELSE
\[
\begin{align*}
&\text{requests}[i][j] \text{ EXCEPT } !.\text{localactive} = \text{false} \\
&\text{ELSE} \\
&\text{requests}[i][j] \text{ EXCEPT } !.\text{localactive} = \text{false} \\
&\text{ELSE} \\
&\text{requests}[i][j] \\
&\text{ELSE} \\
&[j \in 1 . . \text{Len(requests}[i])] \mapsto \text{if } \exists m \in \text{msgs} : \\
&\quad \land \text{Memory} \\
&\quad [\text{proc}] \\
&\quad [\text{array_of_requests} + m] \neq \text{MPI_REQUEST_NULL} \\
&\quad \land \langle i, j \rangle = r(m).\text{match} \\
&\quad \text{THEN} \\
&\quad \text{IF } \neg \text{requests}[i][j].\text{localactive} \\
&\quad \quad \text{THEN} \\
&\quad \quad \text{requests}[i][j] \text{ EXCEPT } !.\text{globalactive} = \text{false} \\
&\quad \quad \text{ELSE} \\
&\quad \quad \text{requests}[i][j] \\
&\quad \quad \text{ELSE} \\
&\quad \quad \text{requests}[i][j]] \\
&\quad \lor \land \forall \ i \in (0 . . (\text{incount} - 1)) : \\
&\quad \quad \lor \land \text{Memory}[\text{proc}][\text{array_of_requests} + i] = \text{MPI_REQUEST_NULL} \\
&\quad \quad \lor \land \text{Memory}[\text{proc}][\text{array_of_requests} + i] \neq \text{MPI_REQUEST_NULL} \\
&\quad \quad \land \neg \text{requests} \\
&\quad \quad [\text{proc}] \\
&\quad \quad [\text{Memory}[\text{proc}][\text{array_of_requests} + i]].\text{localactive} \\
&\quad \quad \land \text{Memory}' = [\text{Memory} \text{ EXCEPT } ![\text{proc}] = \\
&\quad \quad \quad [@ \text{EXCEPT } ![\text{outcount} = \text{MPI_UNDEFINED}]] \\
&\quad \quad \land \text{UNCHANGED } \langle \text{requests} \rangle \\
&\quad \quad \land \text{UNCHANGED } \langle \text{group}, \text{communicator}, \text{bufsize}, \text{message_buffer}, \\
&\quad \quad \quad \text{initialized, collective} \rangle
\end{align*}
\]

Test for some subset of the requests referenced in the \text{array_of_requests} to complete. Defined in terms of \text{MPI_Waitsome}.

\[
\text{MPI_Waitsome}(\text{incount}, \text{array_of_requests}, \text{outcount}, \\
\text{array_of_indices}, \text{array_of_statuses}, \text{return}, \text{proc}) \triangleq \\
\text{LET } r(v) \triangleq \text{requests}[\text{proc}][\text{Memory}[\text{proc}][\text{array_of_requests} + v]] \text{IN} \\
\text{LET } \text{msgs} \triangleq \{x \in (0 . . (\text{incount} - 1)) : \\
\land \text{Memory} \\
\quad [\text{proc}] \\
\quad [\text{array_of_requests} + x] \neq \text{MPI_REQUEST_NULL} \\
\land r(x).\text{localactive} \quad \text{The request is active}.
\land \lor r(x).\text{transmitted} \quad \text{The communication actually happened or}
\]
\[ \forall r(x).\text{canceled} \quad \text{the communication got canceled or} \]
\[ \lor r(x).\text{buffered} \} \quad \text{the communication got buffered} \]
\[ \land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 - 200.12 \]
\[ \text{"Error: MPI\_Testsome called with proc not in initialized state."} \]
\[ \land \forall i \in (0..(\text{incount} - 1)): \quad 49.35 - 49.36, 49.5 \]
\[ \land \text{Memory}[\text{proc}][\text{array\_of\_requests} + i] \neq \text{MPI\_REQUEST\_NULL} \]
\[ \land r(i).\text{localactive} \]
\[ \land \text{IF Cardinality}(\text{msgs}) > 0 \quad \text{number of completed messages} \]
\[ \text{THEN} \]
\[ \land \exists \text{seq} \in [1..\text{Cardinality}(\text{msgs}) \rightarrow \text{msgs}]: \]
\[ \land \forall s \in \text{msgs}: \]
\[ \exists n \in 1..\text{Len(seq)}: \]
\[ \land \text{seq}[n] = s \quad 48.47 - 49.2 \]
\[ \land \forall m \in 1..\text{Len(seq)}: \text{seq}[n] = \text{seq}[m] \Rightarrow m = n \]
\[ \land \text{Memory'} = \]
\[ [\text{Memory EXCEPT ![proc} = \]
\[ [j \in 1..\text{Len(Memory[proc])] \mapsto \]
\[ \text{IF } j \in \text{array\_of\_requests} . . (\text{array\_of\_requests} + (\text{incount} - 1)) \]
\[ \text{THEN} \]
\[ \text{IF } \land \exists m \in \text{msgs} : \land j = (\text{array\_of\_requests} + m) \]
\[ \land \neg r(m).\text{persist} \]
\[ \text{THEN } \text{MPI\_REQUEST\_NULL} \]
\[ \text{ELSE } \text{Memory[proc][j]} \]
\[ \text{ELSE} \]
\[ \text{IF } \land j \in \text{array\_of\_indices} . . \]
\[ (\text{array\_of\_indices} + (\text{incount} - 1)) \]
\[ \land (j - \text{array\_of\_indices}) < \text{Len(seq)} \]
\[ \text{THEN} \]
\[ \text{seq}[j - \text{array\_of\_indices} + 1] \]
\[ \text{ELSE} \]
\[ \text{IF } \land j \in \text{array\_of\_statuses} . . \]
\[ (\text{array\_of\_statuses} + ((\text{incount} \ast 5) - 1)) \]
\[ \land ((j - \text{array\_of\_statuses}) \div 5) < \text{Len(seq)} \]
\[ \text{THEN} \]
\[ \text{LET } \text{offset} \leftarrow j - \text{array\_of\_statuses} \]
\[ \text{LET } \text{reqj} \leftarrow \text{r(seq}[(\text{offset} \div 5) + 1)]) \]
\[ \text{IF } \lor \text{reqj.message.src} = \text{MPI\_PROC\_NULL} \]
\[ \lor \text{reqj.message.dest} = \text{MPI\_PROC\_NULL} \]
\[ \text{THEN } \text{SetStatus}(\text{offset}, \quad 61.3 - 61.4 \]
\[ \text{FALSE}, \]
\[ 0, \]
\[ \text{MPI\_PROC\_NULL}, \]
1669 \textit{MPI\_ANY\_TAG,}
1670 0)
1671 ELSE \textit{SetStatus}\((\text{offset, } 48.17 - 48.18\)
1672 \textit{reqj.canceled} \&\& \neg \textit{reqj.transmitted,}
1673 \textit{reqj.message.numelements,}
1674 \textit{reqj.message.src,}
1675 \textit{reqj.message.msgtag,}
1676 \textit{reqj.error})
1677 ELSE
1678 IF \(j \text{ = } \text{outcount}\)
1679 THEN \textit{Cardinality}\((\text{msgs})\)
1680 ELSE \textit{Memory}\([\text{proc}][j]\\])
1681 \& \textit{requests}' =
1682 \[j \in (0 \ldots (N - 1)) \mapsto\]
1683 IF \(j \text{ = } \text{proc}\)
1684 THEN
1685 \[k \in 1 \ldots \text{Len(}\text{requests}[j]\\)) \mapsto\]
1686 IF \(\exists \text{ } m \in \text{msgs} :\)
1687 \[k = \text{Memory}[j][\text{array\_of\_requests } + m]\]
1688 THEN
1689 IF \textit{requests}[j][k].\textit{match} \neq \langle \rangle
1690 THEN
1691 IF \textit{requests}[requests[j][k].\textit{match}[1]]
1692 \textit{requests}[j][k].\textit{match}[2].\textit{localactive}
1693 THEN
1694 \textit{requests}[j][k] \text{EXCEPT} \textit{!.localactive = FALSE,}
1695 \textit{!.globalactive = FALSE}\]
1696 ELSE
1697 \textit{requests}[j][k] \text{EXCEPT} \textit{!.localactive = FALSE}\]
1698 ELSE
1699 \textit{requests}[j][k] \text{EXCEPT} \textit{!.localactive = FALSE}\]
1700 ELSE
1701 \textit{requests}[j][k]\]
1702 ELSE
1703 \[k \in 1 \ldots \text{Len(}\text{requests}[j]\\)) \mapsto\]
1704 IF \(\exists \text{ } m \in \text{msgs} :\)
1705 \& \textit{Memory} \[\text{proc}\]
1706 \[\text{array\_of\_requests } + m \neq \text{MPI\_REQUEST\_NULL}\]
1707 \& \langle j, k \rangle = r(m).\textit{match}\]
1708 THEN
1709 IF \neg \textit{requests}[j][k].\textit{localactive}
1710 THEN
What happens in the following scenario: 1: send 2: probe 1: cancel 2: recv

Note: the conjunct \( m.universe = comm \) assumes uniqueness across processes and time. This is too strong for a completely general model and should be fixed when the communicators are added.

Probe for a message. Nonblocking; note the leading IF.

\[ MPI_Iprobe(source, tag, comm, flag, status, return, proc) \triangleq \]

\[ \land Assert(initialized[proc] = "initialized", 200.10 - 200.12 \]

"Error: MPI_Testany called with proc not in initialized state."

\[ \land Assert(source \neq MPI_PROC_NULL, \]

"No mention of MPI_PROC_NULL being used with MPI_Iprobe in the standard."

\[ \land IF \exists i \in (0 .. (N - 1)) : \]

\[ \exists j \in (1 .. Len(requests[i])) : \]

LET \( m \triangleq requests[i][j].message\)

\[ \land \lor m.src = source \]

\[ \lor source = MPI\_ANY\_SOURCE \]

\[ \land \lor m.msgtag = tag \]

\[ \lor tag = MPI\_ANY\_TAG \]

\[ \land m.universe = comm \] see note

\[ \land m.state = "send" \] 51.41 - 51.42 must match

\[ \land requests[i][j].globalactive \] 51.41 - 51.42

\[ \land \lnot requests[i][j].transmitted \]
\( \neg \text{requests}[i][j].\text{canceled} \)

\[ \exists i \in (0 \ldots (N - 1)) : \quad 51.39 - 51.41 \]

\[ \exists j \in (1 \ldots \text{Len(requests}[i]) : \quad 51.39 - 51.41 \]

\( \text{LET } m \triangleq \text{requests}[i][j].\text{messageIN} \)

\( \land \lor m.\text{src} = \text{source} \)

\( \lor \text{source} = \text{MPI\_ANY\_SOURCE} \)

\( \land \lor m.\text{msgtag} = \text{tag} \)

\( \lor \text{tag} = \text{MPI\_ANY\_TAG} \)

\( \land \lor m.\text{universe} = \text{comm} \quad \text{see note} \)

\( \land \text{m.state} = \text{"send"} \quad 51.41 - 51.42 \text{ must match} \)

\( \land \text{requests}[i][j].\text{globalactive} \quad 51.41 - 51.42 \)

\( \land \neg \text{requests}[i][j].\text{transmitted} \)

\( \land \neg \text{requests}[i][j].\text{canceled} \)

\( \forall k \in (1 \ldots \text{Len(requests}[i]) : \quad \text{least match} \)

\( \land \text{requests}[i][k].\text{globalactive} \)

\( \land \neg \text{requests}[i][k].\text{canceled} \)

\( \land \neg \text{requests}[i][k].\text{transmitted} \)

\( \Rightarrow j \leq k \)

\( \land \text{Memory'} = [\text{Memory EXCEPT ![proc] = 51.42} \]

\( \quad \text{@ EXCEPT ![\text{Status\_Canceled}(\text{status})] = \text{FALSE},} \)

\( \quad ![\text{Status\_Count}(\text{status})] = m.\text{numelements}, \)

\( \quad ![\text{Status\_Source}(\text{status})] = m.\text{src}, \)

\( \quad ![\text{Status\_Tag}(\text{status})] = m.\text{msgtag}, \)

\( \quad ![\text{Status\_Err}(\text{status})] = \text{requests}[i][j].\text{error}, \)

\( \quad ![\text{flag} = 1] 51.39 \]

\( \text{ELSE} \)

\( \land \text{Memory'} = \)

\( \quad [\text{Memory EXCEPT ![proc] = 51.42} \]

\( \quad \text{@ EXCEPT ![\text{flag} = 0]] 51.44 \text{ Status is undefined} \]

\( \land \text{UNCHANGED } \langle \text{group, communicator, bufsize, message buffer,} \)

\( \quad \text{requests, initialized, collective} \rangle \)

\( \text{Wait on a probe for a message. 52.24 – 52.25} \)

\( \text{MPI\_Probe(source, tag, comm, status, return, proc) } \triangleq \)

\( \land \text{Assert(initialized[proc] = "initialized", 200.10 – 200.12} \)

\( \quad \text{"Error: MPI\_Testany called with proc not in initialized state."} ) \)

\( \land \text{Assert(source } \neq \text{ MPI\_PROC\_NULL, } \)

\( \quad \text{"No mention of MPI\_PROC\_NULL used with MPI\_Probe in the standard."} ) \)

\( \land \exists i \in (0 \ldots (N - 1)) : \)

\( \exists j \in (1 \ldots \text{Len(requests}[i]) : \)

\( \text{LET } m \triangleq \text{requests}[i][j].\text{messageIN} \)

\( \land \lor m.\text{src} = \text{source} \)

\( \lor \text{source} = \text{MPI\_ANY\_SOURCE} \)
\( m \text{.msgtag} = \text{tag} \)
\( \lor \text{tag} = \text{MPI\_ANY\_TAG} \)
\( m \text{.universe} = \text{comm} \) see note for iprobe
\( m \text{.state} = \text{"send"} \)
\( \land \lnot \text{requests}[i][j].\text{transmitted} \)
\( \land \text{requests}[i][j].\text{canceled} \)
\( \forall k \in (1 \ldots \text{Len} (\text{requests}[i])) : \)
\( \land \text{requests}[i][k].\text{globalactive} \)
\( \land \lnot \text{requests}[i][k].\text{transmitted} \)
\( \Rightarrow j \leq k \)
\( \land \text{Memory}' = [\text{Memory} \text{EXCEPT} ![\text{proc}] = 51.42 \]
\( \text{unchanged} \langle \text{group, communicator, bufsize, message_buffer, requests, initialized, collective} \rangle \)

**Cancel an active request.**

What do you do when the request is \text{MPI\_REQUEST\_NULL}?  
\( \land \text{requests}' = [\text{requests} \text{EXCEPT} ![\text{proc}] = 54.8 - 54.10 \]
\( \text{unchanged} \langle \text{group, communicator, bufsize, message_buffer, requests, initialized, collective} \rangle \)

**Test whether a request was canceled successfully.**

\( \land \text{Memory}' = [\text{Memory} \text{EXCEPT} ![\text{proc}] = 54.46 - 55.1 \]
\( \text{unchanged} \langle \text{group, communicator, bufsize, message_buffer, requests, initialized, collective} \rangle \)

---

Section 3.9 Persistent communication requests

Create a persistent standard mode send request.
Create a persistant buffered mode send request.

\[
\text{MPI\_Send\_init}(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq \]
\[
\wedge \text{Assert} \left( \text{initialized}[proc] = \text{"initialized"}, \quad 200.10 - 200.12 \right) \\
\quad \text{"Error: MPI\_Send\_init called with proc not in initialized state."} \\
\wedge \text{requests'} = \left[ \text{requests} \text{ EXCEPT ![proc]} = 56.4 - 56.5 \right] \\
\text{LET } msg \triangleq \left[ \begin{array}{l}
\text{addr} \mapsto buf, \\
\text{src} \mapsto \text{Rank}(proc, comm), \\
\text{dest} \mapsto dest, \\
\text{msgtag} \mapsto tag, \\
\text{dtype} \mapsto datatype, \\
\text{nmelelements} \mapsto count, \\
\text{universe} \mapsto comm, \\
\text{state} \mapsto \text{"send"} \\
\end{array} \right] \\
\text{IN} \\
\text{\@ o } \langle \text{Make\_request}(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{TRUE}, \langle \rangle, \text{FALSE}, msg) \rangle \rangle \\
\wedge \text{Memory'} = \\
\left[ \text{Memory \ EXCEPT ![proc]} = \right. \\
\left[ \text{@ \ EXCEPT ![request]} = \\
\text{Len}(\text{requests}[proc]) + 1 \right] \\
\left. \quad \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer,} \\
\text{initialized, collective} \rangle \right]
\]

\[
\text{MPI\_Bsend\_init}(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq \]
\[
\wedge \text{Assert} \left( \text{initialized}[proc] = \text{"initialized"}, \quad 200.10 - 200.12 \right) \\
\quad \text{"Error: MPI\_Bsend\_init called with proc not in initialized state."} \\
\wedge \text{requests'} = \left[ \text{requests} \text{ EXCEPT ![proc]} = 56.26 \right] \\
\text{LET } msg \triangleq \left[ \begin{array}{l}
\text{addr} \mapsto buf, \\
\text{src} \mapsto \text{Rank}(proc, comm), \\
\text{dest} \mapsto dest, \\
\text{msgtag} \mapsto tag, \\
\text{dtype} \mapsto datatype, \\
\text{nmelelements} \mapsto count, \\
\text{universe} \mapsto comm, \\
\text{state} \mapsto \text{"bsend"} \\
\end{array} \right] \\
\text{IN} \\
\text{\@ o } \langle \text{Make\_request}(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{TRUE}, \langle \rangle, \text{FALSE}, msg) \rangle \rangle \\
\wedge \text{Memory'} = \\
\left[ \text{Memory \ EXCEPT ![proc]} = \right. \\
\left[ \text{@ \ EXCEPT ![request]} = \\
\text{Len}(\text{requests}[proc]) + 1 \right] \\
\left. \quad \text{UNCHANGED } \langle \text{group, communicator, bufsize, message\_buffer,} \\
\text{initialized, collective} \rangle \right]
\[ \land \text{UNCHANGED} \langle \text{group, communicator, bufsize, message\_buffer,} \\
\quad \text{initialized, collective} \rangle \]

Create a persistent synchronous mode send request.
\[
\text{MPI\_Ssend\_init}(\text{buf, count, datatype, dest, tag, comm, request, return, proc}) \triangleq
\land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 - 200.12
\quad \text{"Error: MPI\_Ssend\_init called with proc not in initialized state."})
\land \text{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = 56.46
\text{LET } \text{msg} \triangleq \begin{cases} 
\text{addr} & \mapsto \text{buf}, \\
\text{src} & \mapsto \text{Rank}(\text{proc, comm}), \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{"send"}
\end{cases}
\text{IN}
\quad @ o \langle \text{Make\_request}(0, \text{FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,}
\quad \text{"ssend"}, \text{TRUE, \langle \rangle, FALSE, msg})\rangle]
\land \text{Memory}' =
\quad \begin{cases} 
[\text{Memory} \text{ EXCEPT } ![\text{proc}] = \\
[\@ \text{EXCEPT } ![\text{request}] =
\quad \text{Len}(\text{requests}[\text{proc}]) + 1)]
\land \text{UNCHANGED} \langle \text{group, communicator, bufsize, message\_buffer,} \\
\text{initialized, collective} \rangle \]

Create a persistent ready mode send request.
\[
\text{MPI\_Rsend\_init}(\text{buf, count, datatype, dest, tag, comm, request, return, proc}) \triangleq
\land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 - 200.12
\quad \text{"Error: MPI\_Rsend\_init called with proc not in initialized state."})
\land \text{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = 57.18
\text{LET } \text{msg} \triangleq \begin{cases} 
\text{addr} & \mapsto \text{buf}, \\
\text{src} & \mapsto \text{Rank}(\text{proc, comm}), \\
\text{dest} & \mapsto \text{dest}, \\
\text{msgtag} & \mapsto \text{tag}, \\
\text{dtype} & \mapsto \text{datatype}, \\
\text{numelements} & \mapsto \text{count}, \\
\text{universe} & \mapsto \text{comm}, \\
\text{state} & \mapsto \text{"send"}
\end{cases}
\text{IN}
\quad @ o \langle \text{Make\_request}(0, \text{FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,}
\quad \text{"rsend"}, \text{TRUE, \langle \rangle, FALSE, msg})\rangle]
\land \text{Memory}' =
\quad \begin{cases} 
[\text{Memory} \text{ EXCEPT } ![\text{proc}] = \\
[\@ \text{EXCEPT } ![\text{request}] =
\quad \text{Len}(\text{requests}[\text{proc}]) + 1)]
\land \text{UNCHANGED} \langle \text{group, communicator, bufsize, message\_buffer,} \\
\text{initialized, collective} \rangle \]
Create a persistent receive request.

\[ \text{MPI\_Recv\_init}(buf, \text{count}, \text{datatype}, \text{source}, \text{tag}, \text{comm}, \text{request}, \text{return}, \text{proc}) = \]

\[ \wedge \text{Assert}(\text{initialized}[^{\text{proc}}] = \text{"initialized"}, 200.10 - 200.12) \]

\[ \wedge \text{Memory}[^{\text{proc}}] = \text{\[Memory except ![proc] = \[@ except ![request] = \]
\[\text{Len(requests[^{\text{proc}}]) + 1]]\]
\]

\[ \wedge \text{UNCHANGED} \left(\text{group}, \text{communicator}, \text{bufsize}, \text{message\_buffer}, \text{initialized}, \text{collective}\right) \]

Start a persistent communication.

What happens when a ready mode send is started and then the receive is canceled before the communication has a chance to transmit?

\[ \text{MPI\_Start}(\text{request}, \text{return}, \text{proc}) = \]

\[ \wedge \text{Assert}(\neg \text{req.localactive}, 58.9) \]

\[ \wedge \text{Assert}(\neg \text{req.globalactive}, 58.9) \]

\[ \wedge \text{Assert}(\text{Memory[^{\text{proc}}][request] \neq \text{MPI\_REQUEST\_NULL}, 58.9) \]

\[ \wedge \text{ Assert}(\text{Memory[^{\text{proc}}][request] = \text{MPI\_REQUEST\_NULL}, 58.9) \]
∧ Assert(req.ctype = "rsend") ⇒ 58.10 − 58.11
∧ req.message.dest = MPI.PROC_NULL 61.1
∧ ∃ j ∈ (0 .. (N − 1)):
   ∃ k ∈ (1 .. Len(requests[j])):
       ∧ requests[j][k].globalactive
       ∧ ¬ requests[j][k].transmitted
       ∧ ¬ requests[j][k].canceled
   ∧ Match(req.message, requests[j][k].message),
   "Error: MPI_Start with rsend and no matching message exists."
∧ Assert(req.ctype = "bsend")
   message_buffer[proc] < bufsize[proc],
   "Error: MPI_Start insufficient buffering available for bsend."
∧ Assert(req.persist, 57.44 − 57.45, 58.8
   "Error: MPI_Start tried to start a non-persistant request."
∧ requests' = [requests EXCEPT ![proc] =
   [@ EXCEPT ![Memory[proc][request]] =
   [@ EXCEPT
   !.localactive = TRUE, 58.9
   !.globalactive = TRUE, 58.9
   !.started = TRUE,
   !.transmitted = FALSE,
   !.canceled = FALSE]]]
∧ IF req.ctype = "bsend"
   THEN
   message_buffer' = [message_buffer EXCEPT ![proc] = @ + 1]
   ELSE
   UNCHANGED ⟨message_buffer⟩
∧ UNCHANGED ⟨group, communicator, bufsize, initialized, collective⟩
∧ UNCHANGED ⟨Memory⟩

Start a list of persistant communications.
Can you start many rsends with only one matching receive posted? –maybe yes
Can you start many bsends with only enough buffering for a subset of the sends? –maybe no

MPI_Startall(count, array_of_requests, return, proc) ≜
LET m ≜ \{ x ∈ (0 .. (count − 1)):
   requests
   [proc]
   [Memory[proc][array_of_requests + x]].ctype = "bsend" \}IN
∧ Assert(initialized[proc] = "initialized", 200.10 − 200.12
   "Error: MPI_Startall called with proc not in initialized state."
∧ Assert(∀ i ∈ (0 .. (count − 1)): ¬ requests[proc][array_of_requests[i]].localactive,
   "Error: MPI_Startall called with some request already active."
∧ Assert(∀ i ∈ (0 .. (count − 1)): ¬ requests[proc][array_of_requests[i]].globalactive,
   "Error: MPI_Startall called with some request already active."
\[\begin{align*}
&\land \textbf{Assert}(\forall i \in (0 \ldots (\text{count} - 1)) : \text{array\_of\_requests}[i] \neq \text{MPI\_REQUEST\_NULL}, \\
&\quad \text{"Error: MPI\_Startall called with some request null."}) \\
&\land \textbf{Assert}(\forall i \in (0 \ldots (\text{count} - 1)) : \\
&\quad \text{requests}[\text{proc}][\text{array\_of\_requests}[i]].\text{ctype} = \text{"rsend"} \Rightarrow 58.10 - 58.11 \\
&\quad \exists j \in (0 \ldots (N - 1)) : \\
&\quad \exists k \in (1 \ldots \text{Len}(\text{requests}[j])) : \\
&\quad \land \text{requests}[j][k].\text{globalactive} \\
&\quad \land \neg \text{requests}[j][k].\text{transmitted} \\
&\quad \land \neg \text{requests}[j][k].\text{canceled} \\
&\quad \land \text{Match}(\text{requests}[\text{proc}][\text{array\_of\_requests}[i]].\text{message}, \\
&\quad \text{requests}[j][k].\text{message}), \\
&\quad \text{"Error: MPI\_Start with rsend when no matching message exists."}) \\
&\land \textbf{Assert}(\forall i \in (0 \ldots (\text{count} - 1)) : \\
&\quad \text{requests}[\text{proc}][\text{array\_of\_requests}[i]].\text{ctype} = \text{"bsend"} \Rightarrow \\
&\quad \text{message\_buffer}[\text{proc}] + \text{Cardinality}(m) < \text{bufsize}[\text{proc}], \\
&\quad \text{"Error: MPI\_Start with bsend when insufficient buffering was available."}) \\
&\land \textbf{requests}' = [\text{requests} \text{ EXCEPT } ![\text{proc}] = \\
&\quad [i \in (1 \ldots \text{Len}(\text{requests}[\text{proc}])) \mapsto \\
&\quad \text{IF } \exists j \in (0 \ldots (\text{count} - 1)) : \text{array\_of\_requests}[j] = i \\
&\quad \text{THEN } 58.9 \\
&\quad \text{[requests}[\text{proc}][i] \text{ EXCEPT} \\
&\quad \text{!\text{localactive} = TRUE,} \\
&\quad \text{!\text{globalactive} = TRUE,} \\
&\quad \text{!.\text{started} = TRUE,} \\
&\quad \text{!.\text{transmitted} = FALSE,} \\
&\quad \text{!.\text{canceled} = FALSE} \\
&\quad \text{ELSE} \\
&\quad \text{requests}[\text{proc}][i]]] \\
&\land \text{message\_buffer}' = [\text{message\_buffer} \text{ EXCEPT } ![\text{proc}] = @ + \text{Cardinality}(m)] \\
&\land \text{UNCHANGED } \langle \text{group, communicator, bufsize, initialized, collective} \rangle \\
&\land \text{UNCHANGED } \langle \text{Memory} \rangle \\
\end{align*}\]
MPI_Barrier_init(comm, return, proc) \triangleq
\begin{align*}
\land & \lor \land \text{collective}[\text{communicator}[\text{proc}][\text{comm}].\text{collective}].\text{state} = \text{"vacant"} \\
\land & \text{collective}' = [\text{collective} \ \text{EXCEPT} \ \! [\text{communicator}[\text{proc}][\text{comm}].\text{collective}] = \\
& \quad \left[ \text{@ EXCEPT} \right. \\
& \quad \quad \! .\text{participants} = \text{@} \cup \\{ \text{proc} \}, \\
& \quad \quad \! .\text{type} = \text{"barrier"}, \\
& \quad \quad \! .\text{state} = \text{"in"} \left] \right. \\
\lor & \land \text{collective}[\text{communicator}[\text{proc}][\text{comm}].\text{collective}].\text{state} = \text{"in"} \\
\land & \text{collective}' = [\text{collective} \ \text{EXCEPT} \ \! [\text{communicator}[\text{proc}][\text{comm}].\text{collective}] = \\
& \quad \left[ \text{@ EXCEPT} \right. \\
& \quad \quad \! .\text{participants} = \text{@} \cup \\{ \text{proc} \}] \\
\land & \text{UNCHANGED} \langle \text{group, communicator, bufsize, message_buffer, requests, initialized, Memory} \rangle
\end{align*}
\[ Memory' = [\text{Memory EXCEPT ![proc] =} \]
\[ [\text{[except ![size] = group[proc][gr].size]}] \]
\[ \land \text{UNCHANGED (group, communicator, bufsize, message_buffer, requests,} \]
\[ \text{initialized, collective)} \]

\[ MPI\_Group\_rank(gr, rank, return, proc) \triangleq \]
\[ \land \text{Assert(initialized[proc] = "initialized", 200.10 – 200.12} \]
\[ \text{"Error: MPI\_Group\_rank called with proc not in initialized state."} \)
\[ \land \text{Memory'} = \]
\[ [\text{Memory EXCEPT ![proc] =} \]
\[ [\text{[except ![rank] =} \]
\[ \begin{align*}
\text{IF proc} & \in \text{group[proc][gr].members} \\
\text{THEN} & \text{group.ranking[proc]} \\
\text{ELSE} & \text{MPI\_UNDEFINED}]
\end{align*} \]
\[ \land \text{UNCHANGED (group, communicator, bufsize, message_buffer, requests,} \]
\[ \text{initialized, collective)} \]

\[ MPI\_Group\_translate\_ranks(group1, n, ranks1, group2, ranks2, return, proc) \triangleq \]
\[ \land \text{Assert(initialized[proc] = "initialized", 200.10 – 200.12} \]
\[ \text{"Error: MPI\_Group\_translate\_ranks called before MPI\_Init."} \)
\[ \land \text{group1} \in \text{MPI\_COMM\_WORLD . .} \]
\[ \text{(MPI\_COMM\_WORLD + MAX\_GROUP)}, \]
\[ \text{"Error: MPI\_Group\_translate\_ranks called with invalid handle for group1."} \)
\[ \land \text{group2} \in \text{MPI\_COMM\_WORLD . .} \]
\[ \text{(MPI\_COMM\_WORLD + MAX\_GROUP),} \]
\[ \text{"Error: MPI\_Group\_translate\_ranks called with invalid handle for group2."} \)
\[ \land \text{n} = \text{Cardinality(DOMAIN ranks1), 138.3} \]
\[ \text{"Error: MPI\_Group\_translate\_ranks called with invalid n."} \)
\[ \land \text{Memory'} = [\text{Memory EXCEPT ![proc] =} \]
\[ [\text{[i} \in 1 .. \text{Len(Memory[proc]) \leftrightarrow} \]
\[ \text{IF i} \in \text{ranks2 . . (ranks2 + n)} \]
\[ \text{THEN} \text{group[proc][group2]} \]
\[ \text{.ranking[group[proc][group1]} \]
\[ \text{.invranking[ranks1[i]]]} \]
\[ \text{ELSE} \text{Memory[proc][i]} \]] \text{not quite right as there is no} \]
\[ \text{possibility of MPI\_UNDEFINED.} \]
\[ \land \text{UNCHANGED mpi\_vars} \]

\[ MPI\_Group\_compare(group1, group2, result, return, proc) \triangleq \]
\[ \land \text{Assert(initialized[proc] = "initialized", 200.10 – 200.12} \]
\[ \text{"Error: MPI\_Group\_compare called before MPI\_Init."} \)
\[ \land \text{Memory'} = \]
\[ [\text{Memory EXCEPT ![proc] =} \]
\[ \emptyset \ \text{EXCEPT} \ [\text{result}] \ = \]
\[ \text{IF} \ \forall \ \text{group1} = \text{group2} \text{ THEN} \]
\[ \forall \ \land \ \text{group}[\text{proc}][\text{group1}].\text{members} = \text{group}[\text{proc}][\text{group2}].\text{members} \]
\[ \land \ \text{group}[\text{proc}][\text{group1}].\text{ranking} = \text{group}[\text{proc}][\text{group2}].\text{ranking} \]
\[ \text{MPI\_IDENT} \]
\[ \text{ELSE} \]
\[ \text{IF} \ \land \ \text{group}[\text{proc}][\text{group1}].\text{members} = \text{group}[\text{proc}][\text{group2}].\text{members} \]
\[ \land \ \text{group}[\text{proc}][\text{group1}].\text{ranking} \neq \text{group}[\text{proc}][\text{group2}].\text{ranking} \]
\[ \text{THEN} \]
\[ \text{MPI\_SIMILAR} \]
\[ \text{ELSE} \]
\[ \text{MPI\_UNEQUAL} \]
size → \text{Cardinality}(\text{group1.members} \cup \text{group2.members}),
ranking → f,
\text{invranking} → \text{finv}]]
\land \text{Memory}' =
\begin{align*}
&M\text{[Memory EXCEPT ![proc] =} \\
&\text{[@ EXCEPT ![newgroup] = } \text{Cardinality(DOMAIN group[proc]) + 1]} \\
&\text{]} \land \text{UNCHANGED } \langle \text{communicator, bufsize, message_buffer,} \\
&\text{requests, initialized, collective} \rangle
\end{align*}

\textit{Section 5.4.1 Communicator Accessors}

\textit{MPI Comm.size}(\text{comm, size, return, proc}) \triangleq
\land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 - 200.12
\text{"MPI_Comm_size called with proc not in initialized state."})
\land \text{Memory}' =
\begin{align*}
&M\text{[Memory EXCEPT ![proc] =} \\
&\text{[@ EXCEPT ![size] =} \\
&\text{group[proc][communicator[proc][comm].group].size]} \\
&\text{]} \land \text{UNCHANGED mpi_vars}
\end{align*}

\textit{MPI Comm.rank}(\text{comm, rank, return, proc}) \triangleq
\land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 - 200.12
\text{"MPI_Comm_rank called with proc not in initialized state."})
\land \text{Memory}' =
\begin{align*}
&M\text{[Memory EXCEPT ![proc] =} \\
&\text{[@ EXCEPT ![rank] = } \text{Rank(\text{proc, comm})]} \\
&\text{]} \land \text{UNCHANGED } \langle \text{group, communicator, bufsize, message_buffer, requests,}
\text{initialized, collective} \rangle
\end{align*}

\textit{MPI Comm.compare}(\text{comm1, comm2, result, return, proc}) \triangleq
\land \text{Assert}(\text{initialized}[\text{proc}] = \text{"initialized"}, \quad 200.10 - 200.12
\text{"MPI_Comm_rank called with proc not in initialized state."})
\land \text{Memory}' =
\begin{align*}
&M\text{[Memory EXCEPT ![proc] =} \\
&\text{[@ EXCEPT ![result] =} \\
&\text{IF } \text{comm1} = \text{comm2} \\
&\text{THEN} \\
&\text{MPI IDENT} \\
&\text{ELSE} \\
&\text{IF } \land \text{communicator[proc][comm1].group} = \\
&\text{communicator[proc][comm2].group} \\
&\land \text{communicator[proc][comm1].group.ranking} = \\
&\text{communicator[proc][comm2].group.ranking} \\
&\text{THEN} \\
&\text{MPI CONGRUENT}
\end{align*}
ELSE
IF communicator[proc][comm1].group =
communicator[proc][comm2].group
THEN
MPI_SIMILAR
ELSE
MPI_UNEQUAL]]
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, 
requests, initialized, collective⟩
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, 
requests, initialized, collective⟩

Section 7.5 Startup

199.12 – 199.17
Initialize the participation of this process within a distributed computation.
MPI_Init(argc, argv, return, proc) ≜
∧ Assert (initialized[proc] = "uninitialized", 199.12
"MPI_Init called with proc not in uninitialized state.”)
∧ initialized' = [initialized EXCEPT ![proc] = "initialized"] 199.13
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, 
requests, collective, Memory⟩
Finalize the participation of this process within a distributed computation.
MPI_Finalize(return, proc) ≜
"Error: MPI_Finalize called with proc not in initialized state.”)
∧ Assert (∀ i ∈ (1 . . Len(requests[proc])) : 199.47
¬requests[proc][i].localactive,
"Error: MPI_Finalize called when some message was still active.”)
∧ Assert (bufsize[proc] = 0, 200.1
"Error: MPI_Finalize called before the buffer is detached.”)
∧ initialized' = [initialized EXCEPT ![proc] = "finalized"] 199.46
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, 
requests, collective, Memory⟩
Determine whether MPI_Init has been called.
MPI_INITIALIZED(flag, return, proc) ≜
∧ Memory' = [Memory EXCEPT ![proc] = ![flag] =
IF initialized[proc] = "initialized" 200.2
THEN 1
ELSE 0]]
∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, requests, 
initialized, collective⟩
“Best effort to clean up”
MPI Abort\((comm,\ errorcode,\ return,\ proc)\) \triangleq
\[
\forall \ p \in (0 \ldots (N - 1)):
\forall \ m \in (1 \ldots \text{Len}(\text{requests}[p])):
\land \ \text{requests}[p][m].\text{globalactive}
\land \neg \text{requests}[p][m].\text{transmitted}
\Rightarrow \text{requests}[p][m]' = [\text{requests}[p][m] \text{ EXCEPT } !.\text{canceled} = \text{TRUE}]
\land \ \text{UNCHANGED} \ (\text{group, communicator, bufsize, message_buffer, requests, initialized, collective})
APPENDIX B

AN EXAMPLE MPI PROGRAM
MODELED IN TLA+

The following model is created from the program shown in Figure 1.2. The model was parsed and simplified to SSA form using the Phoenix framework. An intermediate representation in the MPIC IR of Chapter 4 was created from this simplified form. The model was then inlined, cleaned, and sliced. The shown model is the resultant TLA+ output.
module function
3 extends mpi_base
5 VARIABLE
6 Map, pc
8 program_vars \triangleq ⟨Memory, Map, pc⟩
10 Labels \triangleq \{ "_main: _main", "_main: $L7", "_main: $L6", "_main: $L10",
13 "_main: $L1", "_main: $L2", "_main: $L5", "_main: $L15",
14 "_main: $L16", "_main: $L17", "_main: $L18", "_main: $L19",
17 "_main: $L31", "_main: $L32" \}
19 The initial state of the program model.
25 Program_Variables_Init \triangleq 
26 \& Memory = [i \in 0..(N-1) \mapsto
27 \langle 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle]
28 \& Map = [
29 \_argv \mapsto 1,
30 \_argc \mapsto 2,
31 \_req3 \mapsto 3,
32 \_data1 \mapsto 4,
33 \_stat \mapsto 5,
34 \_data3 \mapsto 10,
35 \_req2 \mapsto 11,
36 \_size \mapsto 12,
37 \_data2 \mapsto 13,
38 \_req1 \mapsto 14,
39 \_rank \mapsto 15,
40 \_flag \mapsto 16,
41 \text{t285} \mapsto 17,
42 \text{t290} \mapsto 18,
43 \text{t295} \mapsto 19]
44 \& pc = [i \in 0..(N-1) \mapsto "_main: _main"]
48 Program_Type_Invariant \triangleq
The final program state of the model (i.e., the return transition).

\[ \text{Final\_State} \triangleq \text{"main: $L32"} \]

An operator to allocate memory appropriately.

\[ \text{AllocateMemory}(\text{ptr}, \text{pid}, \text{size}) \triangleq \]
\[ \land \text{Memory}' = [i \in 0 \ldots (N - 1) \mapsto \]
\[ \quad \text{IF } i = \text{pid} \]
\[ \quad \text{THEN } [j \in 1 \ldots (\text{Len(Memory}[i])] + \text{size} \mapsto \]
\[ \quad \quad \text{IF } j \leq \text{Len(Memory}[i]) \]
\[ \quad \quad \text{THEN} \]
\[ \quad \quad \quad \text{IF } \text{ptr} = j \]
\[ \quad \quad \quad \text{THEN } \text{Len(Memory}[\text{pid}]]) + 1 \]
\[ \quad \quad \quad \text{ELSE } \text{Memory}[i][j] \]
\[ \quad \quad \quad \text{ELSE } \text{"uninitialized memory space"} \]
\[ \quad \quad \text{ELSE } \text{Memory}[i] \]
\[ \text{The program model to be verified.} \]

\[ \text{Proc}(\text{pid}) \triangleq \]
\[ \lor \land \text{pc}[\text{pid}] = \text{"main: _main"} \]
\[ \land \text{pc}' = [\text{pc} \text{ EXCEPT ![pid]} = \text{"main: $L1"}] \]
\[ \land \text{MPI}_\text{-Init}(\text{Map}_\text{-arve}, \text{Map}_\text{-argv}, 0, \text{pid}) \]
\[ \land \text{UNCHANGED } \langle \text{Map} \rangle \]
\[ \lor \land \text{pc}[\text{pid}] = \text{"main: $L7"} \]
\[ \land \text{pc}' = [\text{pc} \text{ EXCEPT ![pid]} = \text{"main: $L16"}] \]
\[ \land \text{Memory}' = [\text{Memory} \text{ EXCEPT ![pid]} = [@ \text{ EXCEPT ![Map}_\text{-data1}} = 0)] \]
\[ \land \text{UNCHANGED } \langle \text{Map} \rangle \]
\[ \land \text{UNCHANGED mpi-vars} \]
\[ \land \text{UNCHANGED mpi-vars} \]
\[ \lor \land \text{pc}[\text{pid}] = \text{"main: $L10"} \]
\[ \land \text{pc}' = [\text{pc} \text{ EXCEPT ![pid]} = \text{"main: $L25"}] \]
\[ \land \text{MPI}_\text{-Wait}(\text{Map}_\text{-req1}, \text{Map}_\text{-stat}, 0, \text{pid}) \]
\[ \land \text{UNCHANGED } \langle \text{Map} \rangle \]
\[ \land \text{UNCHANGED mpi-vars} \]
\[ \land \text{UNCHANGED mpi-vars} \]
\[ \lor \land \text{pc}[\text{pid}] = \text{"main: $L9"} \]
\[ \land \text{pc}' = [\text{pc} \text{ EXCEPT ![pid]} = \text{"main: $L28"}] \]
\[ \land \text{MPI}_\text{-Wait}(\text{Map}_\text{-req2}, \text{Map}_\text{-stat}, 0, \text{pid}) \]
\[ \land \text{UNCHANGED } \langle \text{Map} \rangle \]
\begin{align*}
\forall \land pc[pid] &= \text{"main: $L13$"} \\
\land pc' &= [pc \text{ EXCEPT } ![pid] = \text{"main: $L31$"}] \\
\land MPI_{-}\text{Wait}(Map_{-}\text{req1}, Map_{-}\text{stat}, 0, \text{pid}) \\
\land \text{UNCHANGED } \langle Map \rangle \\
\forall \land pc[pid] &= \text{"main: $L12$"} \\
\land pc' &= [pc \text{ EXCEPT } ![pid] = \text{"main: $L31$"}] \\
\land MPI_{-}\text{Wait}(Map_{-}\text{req2}, Map_{-}\text{stat}, 0, \text{pid}) \\
\land \text{UNCHANGED } \langle Map \rangle \\
\forall \land pc[pid] &= \text{"main: $L1$"} \\
\land pc' &= [pc \text{ EXCEPT } ![pid] = \text{"main: $L2$"}] \\
\land MPI_{-}\text{Comm}_{-}\text{rank}(1140850688, Map_{-}\text{rank}, 0, \text{pid}) \\
\land \text{UNCHANGED } \langle Map \rangle \\
\forall \land pc[pid] &= \text{"main: $L5$"} \\
\land pc' &= [pc \text{ EXCEPT } ![pid] = \text{"main: $L15$"}] \\
\land Memory' &= [\text{Memory EXCEPT } ![\text{pid}] = \\
&\quad [\@ \text{ EXCEPT } ![\text{Map}.t285] = \text{Memory}[\text{pid}][\text{Map}_{-}\text{rank}] = 0]] \\
\land \text{UNCHANGED } \langle Map \rangle \\
\land \text{UNCHANGED } mpi_{-}\text{vars} \\
\forall \land pc[pid] &= \text{"main: $L15$"} \\
\land pc' &= [pc \text{ EXCEPT } ![\text{pid}] = \text{"main: $L7$"}] \\
\land Memory[\text{pid}][\text{Map}.t285] \\
\land \text{UNCHANGED } \langle Map, Memory \rangle \\
\land \text{UNCHANGED } mpi_{-}\text{vars} \\
\forall \land pc[pid] &= \text{"main: $L16$"} \\
\land pc' &= [pc \text{ EXCEPT } ![\text{pid}] = \text{"main: $L17$"}] \\
\land Memory' &= [\text{Memory EXCEPT } ![\text{pid}] = \@ \text{ EXCEPT } ![\text{Map}_{-}\text{data2}] = 0]] \\
\land \text{UNCHANGED } \langle Map \rangle \\
\land \text{UNCHANGED } mpi_{-}\text{vars} \\
\forall \land pc[pid] &= \text{"main: $L17$"} \\
\land pc' &= [pc \text{ EXCEPT } ![\text{pid}] = \text{"main: $L18$"}] \\
\land MPI_{-}\text{Irecv}(Map_{-}\text{data1}, 1, 1275069445, 1, 0, \\
&\quad 1140850688, Map_{-}\text{req1}, 0, \text{pid}) \\
\land \text{UNCHANGED } \langle Map \rangle \\
\forall \land pc[pid] &= \text{"main: $L18$"}
\end{align*}
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L19"} ] \]
\[ MPI\_\text{Irecv}(\text{Map.\_data2}, 1, 1275069445, 1, 1, \]
\[ 1140850688, \text{Map.\_req2}, 0, \text{pid}) \]
\[ \text{UNCHANGED} \langle \text{Map} \rangle \]
\[ V \land pc[\text{pid}] = "\text{main: $L19"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L23"} ] \]
\[ MPI\_\text{Irecv}(\text{Map.\_data3}, 1, 1275069445, 1, 2, \]
\[ 1140850688, \text{Map.\_req3}, 0, \text{pid}) \]
\[ \text{UNCHANGED} \langle \text{Map} \rangle \]
\[ V \land pc[\text{pid}] = "\text{main: $L21"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L22"} ] \]
\[ Memory' = [\text{Memory except } !\text{[pid]} = [@ \text{except } !\text{[Map.\_data2]} = 6]] \]
\[ \text{UNCHANGED} \langle \text{Map} \rangle \]
\[ \text{UNCHANGED mpi\_vars} \]
\[ V \land pc[\text{pid}] = "\text{main: $L22"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L23"} ] \]
\[ MPI\_\text{Issend}(\text{Map.\_data1}, 1, 1275069445, 0, 1, \]
\[ 1140850688, \text{Map.\_req1}, 0, \text{pid}) \]
\[ \text{UNCHANGED} \langle \text{Map} \rangle \]
\[ V \land pc[\text{pid}] = "\text{main: $L23"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L24"} ] \]
\[ Memory' = [\text{Memory except } !\text{[pid]} = \]
\[ [@ \text{except } ![\text{Map.t290}] = \text{Memory}[\text{pid}][\text{Map.\_rank}] = 1]] \]
\[ \text{UNCHANGED} \langle \text{Map} \rangle \]
\[ \text{UNCHANGED mpi\_vars} \]
\[ V \land pc[\text{pid}] = "\text{main: $L24"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L10"} ] \]
\[ Memory[\text{pid}][\text{Map.t290}] \]
\[ \text{UNCHANGED} \langle \text{Map, Memory} \rangle \]
\[ \text{UNCHANGED mpi\_vars} \]
\[ V \land pc[\text{pid}] = "\text{main: $L24"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L9"} ] \]
\[ \neg(Memory[\text{pid}][\text{Map.t290}]) \]
\[ \text{UNCHANGED} \langle \text{Map, Memory} \rangle \]
\[ \text{UNCHANGED mpi\_vars} \]
\[ V \land pc[\text{pid}] = "\text{main: $L25"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L26"} ] \]
\[ MPI\_\text{Irsend}(\text{Map.\_data2}, 1, 1275069445, 0, 0, \]
\[ 1140850688, \text{Map.\_req2}, 0, \text{pid}) \]
\[ \text{UNCHANGED} \langle \text{Map} \rangle \]
\[ V \land pc[\text{pid}] = "\text{main: $L26"} \]
\[ pc' = [pc \text{ except } !\text{[pid]} = "\text{main: $L28"} ] \]
\[ MPI\_\text{Irsend}(\text{Map.\_data3}, 1, 1275069445, 0, 2, \]
∧ UNCHANGED \langle Map \rangle \\
∨ ∧ pc[\text{pid}] = "\text{main: }$L28"\\n∧ pc' = [pc \text{ EXCEPT } ![\text{pid}] = "\text{main: }$L29"]\\n∧ Memory' = [Memory \text{ EXCEPT } ![\text{pid}] = \\
[\text{@ EXCEPT } ![Map.t295] = Memory[\text{pid}][Map._\text{rank}] = 0]\\n∧ UNCHANGED \langle Map \rangle \\
∧ UNCHANGED mpi_vars \\
∨ ∧ pc[\text{pid}] = "\text{main: }$L29"\\n∧ pc' = [pc \text{ EXCEPT } ![\text{pid}] = "\text{main: }$L13"]\\n∧ Memory[\text{pid}][Map.t295]\\n∧ UNCHANGED \langle Map, Memory \rangle \\
∧ UNCHANGED mpi_vars \\
∨ ∧ pc[\text{pid}] = "\text{main: }$L29"\\n∧ pc' = [pc \text{ EXCEPT } ![\text{pid}] = "\text{main: }$L12"]\\n∧ \neg (Memory[\text{pid}][Map.t295])\\n∧ UNCHANGED \langle Map, Memory \rangle \\
∧ UNCHANGED mpi_vars \\
∨ ∧ pc[\text{pid}] = "\text{main: }$L31"\\n∧ pc' = [pc \text{ EXCEPT } ![\text{pid}] = "\text{main: }$L32"]\\n∧ MPI\_Finalize(0, \text{pid})\\n∧ UNCHANGED \langle Map \rangle 
APPENDIX C

THE TOP LEVEL TLA+ SPECIFICATION

This appendix includes the remaining TLA+ specification pieces necessary to use TLC in connection with the model checking framework described in this dissertation.
module MPI_Program

Specification to verify.
Robert Palmer
The University of Utah
School of Computing

EXTENDS function

The variables used in the specification.
vars \(\triangleq\) \langle group, communicator, bufsize, message_buffer, requests, initialized, collective, pc, Memory, Map \rangle

Initial values for all variables in the model.
Init \(\triangleq\)
\& Program_Variables_Init
\& MPI_Specification_Init

The next state relation defined by the program being verified.
Next \(\triangleq\)
\lor \& Pair
\& UNCHANGED program_vars
\lor \& Transmit
\& UNCHANGED \langle Map, pc \rangle
\lor \& Buffer
\& UNCHANGED program_vars
\lor \exists p \in 0..(N-1) : Proc(p)
\lor \forall p \in (0..(N-1)) :
\& pc[p] = Final_State
\& initialized[p] = "finalized"
\& UNCHANGED program_vars
\& UNCHANGED mpi_vars

A special termination condition that says eventually all processes reach the end label, \textit{i.e.}, all processes eventually exit.
Termination \(\triangleq\) \& (\forall p \in (0..(N-1)) : pc[p] = Final_State)

The specification to verify via TLC.
Spec \(\triangleq\) \& Init
\& \Box [Next] vars
\&Termination

The type invariant
\( \text{Type}_\text{Invariant} \equiv \)
\( \land \text{MPI}_\text{Type}_\text{Invariant} \)
\( \land \text{Program}_\text{Type}_\text{Invariant} \)
REFERENCES


transaction-based reduction without cycle detection. In Patrice Godefroid,
editor, *Model Checking Software: 12th International SPIN Workshop*, volume


In *MPC ’98: Proceedings of the Mathematics of Program Construction*, pages

Petri nets: applications and relationships to other models of concurrency*, pages


[57] Microsoft. Phoenix academic program

[58] Microsoft. Static Driver Verifier (SDV)

[59] Madanlal Musuvathi and David L. Dill. An incremental heap canonicalization

[60] Ratan Nalumasu and Ganesh Gopalakrishnan. Pv: An explicit enumeration

[61] Ratan Nalumasu and Ganesh Gopalakrishnan. An efficient partial order reduc-
tion algorithm with an alternative proviso implementation. *Formal Methods

Cil: Intermediate language and tools for analysis and transformation of c


