Practical Formal Verification of MPI and Thread Programs

Tutorial Notes

Ganesh Gopalakrishnan

School of Computing, University of Utah, Salt Lake City, UT 84112

http://www.cs.utah.edu/formal_verification/europvm09-tutorial-mpi-threading-fv

http://www.cs.utah.edu/formal_verification/ISP-release

1 Introduction

This document is being prepared to serve as a reference for the tutorial with the same title that will be offered during EuroPVM/MPI 2009 in Espoo, September 7, 2009. In this tutorial, we will mostly focus on the topic of formal *dynamic verification* [1] of MPI programs using our tool ISP. There will also be a short session on shared memory program verification using our tool Inspect. Section 4 discusses Inspect/threading; until this section, this document focuses on ISP and MPI dynamic formal verification. In this tutorial document, we assume familiarity with MPI and basic threading concepts. In the EuroPVM/MPI tutorial, we will provide short introductions to MPI and threading.

ISP (In-Situ Partial order) is our dynamic formal verification tool for MPI programs. Currently, ISP can verify MPI C programs (Fortran porting is underway), and employs available standard MPI libraries to provide the semantics of MPI calls. So far, ISP has been tested successfully using the MPI libraries MPICH2, OpenMPI, and Microsoft MPI. To a lesser extent, ISP has also been tested using MVAPICH and IBM MPI. ISP has been successfully run on the Linux, Mac OS/X, and Windows operating systems. As to graphical user interfaces, ISP has two of them, one created using Visual Studio and the other using Eclipse. This document is written without specific references to the particular high level language, MPI library, operating system, or GUI used by ISP. It is meant to cover the basic principles underlying all these versions of ISP. Specific details pertaining to these platforms, including user manuals, are available on the ISP website.

With these preliminaries out of the way, we are now ready to present the heart of ISP – namely, its central algorithm POE (Partial Order avoiding Elusive interleavings). We name the algorithm employed by ISP separately because in future we plan to have other algorithms in lieu of POE (the tool name is expected to remain ISP). Let us study POE using the example executions shown in Figure 1. In the pseudo-code presented in this figure, we have suppressed details of the data shipped by the communication commands, as well as computation oriented statements that would typically be employed between MPI calls. Also we employ $\star$ to highlight *wildcard receive*.

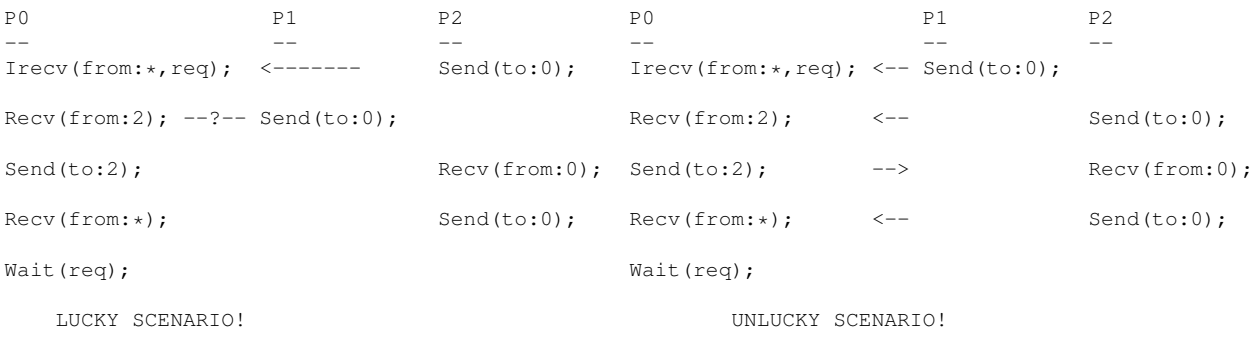


Figure 1: Two execution scenarios of a simple MPI program

In the run entitled `LUCKY SCENARIO`, because of the timings in the system, `P2`'s `Send(to:0)` matches with `P0`'s `Irecv(from:*,req)`. However, this will result in a deadlock as shown by the actions connected by `--?--` which cannot match, and hence the execution freezes at this point. This execution is called 'lucky' because a test engineer would detect the deadlock during testing if this scenario were to manifest during testing. But suppose the execution shown on the right-hand side entitled `UNLUCKY SCENARIO` is all that emerges during testing; the test engineer could then consider the program to be correct and ship it. A customer who uses the program may, however, encounter the first behavior (not so 'lucky' for him/her)!

This simple example illustrates the futility of testing concurrent programs in general, and MPI programs in particular. One simply has no idea what the communication delays on a machine are, or the scheduling policies used by the MPI runtime or the underlying operating system. One popular approach to increase the changes of bug detection is to perturb schedules by throwing in 'random sleep' statements into the code. While this method can often help manifest previously untried schedules (interleavings), it certainly provides no guarantees of covering *all relevant* schedules, or of covering each interleaving exactly once. Besides, the delays introduced can slow down the whole testing process. These are the deficiencies that POE overcomes.

In a nutshell, ISP's verification attempts to simulate *all possible speed/scheduling/machine variations* in one fell swoop. ISP achieves this by employing the architecture shown in Figure 2. The MPI actions issued by the executable are shown as `MPI_f` for some MPI function `f`. These functions are intercepted, and issued as `PMPI_f` into the MPI runtime, but only as and when dictacted by the POE algorithm. The POE algorithm opportunistically rearranges the sequence of `MPI_f` calls seen to a different sequence of `PMPI_f` calls. It also alters some of the functions `f` sent into the runtime. All this is done to pursue the set of all *relevant interleavings*. In other words, POE *de-biases* from the specific delays or scheduling policies of the platform on which verification using ISP is being conducted such that when a program is shown correct by ISP, it remains correct *no matter what the MPI library or platform* that the customer may have! We now discuss these ideas beginning with the notion of *relevant interleavings*.

1.1 Relevant Interleavings

So what are the relevant interleavings to be executed in an MPI program? In the example in Figure 1, the relevant interleavings are both the `LUCKY` and the `UNLUCKY` executions. In the example shown in Figure 3 also, there are two relevant interleavings, one for each `Isend` match with the wildcard `Irecv`.

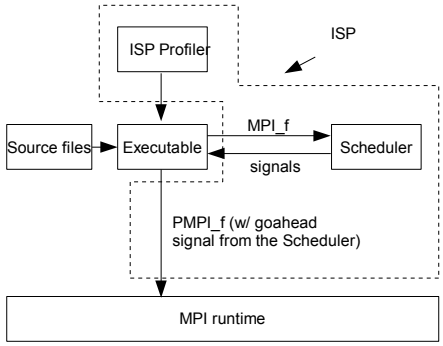


Figure 2: Overview of ISP

b

```
P0: Isend(to:1, data = 42) ; ...
P1: Irecv(from:*, x)              ; if (x==42) then error1 else ...
P2: Isend(to:1, data = 21) ; ...
```

Figure 3: Simple MPI Example Illustrating Wildcard Receives

This example shows that depending on the non-deterministic wildcard match, the control flow following the `Irecv` can vary. Specifically, the error `error1` is triggered depending on the `Isend` that matches the wildcard receive `Irecv`. ISP's approach to generate all relevant interleavings is to track all non-deterministic constructs, and ensure that each of them is examined in the most general ("schedule de-biased") manner.

While all this is a special case of the well known[1] *partial order reduction* (POR) approach [2, Chapter 10], ISP's approach to POR is considerably different from that taken in the case of classical shared memory in-order execution languages. The out-of-order execution semantics, together with the rich set of collective operations that MPI uses results in POE being dramatically different from traditional POR algorithms, as explained in the sections to follow. Overall, POE's execution method is quite different from those used in all dynamic verification tools we know of.

1.2 Dynamic Rewriting and Reordering

In order to generate all relevant schedules, it is not sufficient to control the issue order of MPI calls. For instance, in Figure 3, it is not sufficient to issue the `Isend` of `P0` and that of `P2` in a certain order. The MPI runtime will, most likely, not heed this issue order and internally re-arrange things so that `P0`'s `Isend` (for example) always ends up matching `P1`'s `Irecv`. The general idea is that ISP's profiler and scheduler (Figure 2) trap MPI commands, and issue these commands into the MPI run-time "when convenient," or "only when forced."

Two crucial aspects of ISP are dynamic re-ordering and dynamic re-writing. Even though the MPI calls pour into the scheduler according to the *program order* of the MPI processes, the scheduler may – if warranted – permute this order into the *internal issue order*. Such reorderings will be illustrated in Section 1.3. Also, when issuing the MPI calls, the scheduler may also choose to *dynamically replace* the arguments of various calls. In the example of Figure 1, the scheduler will replace `Irecv(from:*,req)` with `Irecv(from:1,req)` during one interleaving. This will create the `UNLUCKY SCENARIO`. It will then re-execute the MPI program, but now replace `Irecv(from:*,req)` with `Irecv(from:2,req)`, thus forcing the `LUCKY SCENARIO`. The same kind of rewriting will also ensure that `error1` of Figure 3 will be caught when `Irecv(from:*)` is replaced by `Irecv(from:0)` during one of the replays, and the value of `42` flows into `x`.

1.3 Handling Barriers and Collectives

According to the MPI library semantics, no MPI process can issue an instruction past its barrier unless all other processes have issued their barrier calls. This does *not* mean that the operations issued *before* the barrier have "finished" (what 'finished' means will be defined later). Therefore, an MPI program must be designed in such a way that when an MPI process reaches a barrier call, all other MPI processes also reach their barrier calls (in the MPI parlance, these are *collective operations*); a failure to do so deadlocks

[1] If you don't know about POR, please do not worry; please skip all such remarks intended at those who do.

```
P0: Isend0(to:1, h0)  ; Barrier0 ; Wait(h0)
P1: Irecv(from:*, h1) ; Barrier1 ; Wait(h1)
P2:                     Barrier2 ; Isend2(to:1, h2) ; Wait(h2)
```

Figure 4: Illustration of Barrier Semantics and the POE Algorithm

the execution. While these rules match the rules followed by other languages and libraries in supporting their barrier operations, in the case of MPI, it is possible for a process P_i to have a non-blocking operation OP *before* its barrier call, and another process P_j to have an operation OP' after P_j's matching barrier call where OP can observe the execution of OP'. This means that OP can, in effect, complete after P_i's barrier has been invoked. Such behaviors are allowed in MPI to keep it a high-performance API.

Figure 4 illustrates the scenario just discussed. In this example, one `Isend` issued by `P0`, shown as `Isend0(to:1, &h0)`, and another issued by `P2`, shown as `Isend2(to:1, &h2)`, target a wildcard `Irecv` issued by `P1`, shown as `Irecv(from:*, h1)`. The following execution is possible: (i) `Isend0(to:1, &h0)` is issued, (ii) `Irecv(from:*, h1)` is issued, (iii) each process *fully* executes its own barrier, and this "collective operation" finishes, (iv) `Isend2(to:1, h2)` is issued, (v) now both sends and the receive are alive, and hence `Isend0` and `Isend2` become dependent (non-deterministic matches with `Irecv`), requiring a dynamic algorithm to pursue both matches. Notice that `Isend0` can finish before `Barrier0` and `Irecv` can finish after `Barrier1`. We sometimes refer to the placement of barriers as in Figure 4 as "*crooked barriers*," because the barriers are used even though there are instructions around it that observe each other.

To handle this example properly, the POE algorithm of ISP cannot assume that `Isend0` is the only sender that can match `Irecv`. Therefore ISP must somehow delay issuing `Irecv` into the MPI run-time until all its matchers are known. In this example, ISP will collect `Isend0` and `Irecv`, then issue all the `Barrier`s, then encounter `Isend2`, and *then only act on* `R`. Here, ISP will issue `Irecv` with arguments `0` and `2` respectively in two consecutive replays of the whole program. Thus, the inner workings of ISP's POE algorithm are as follows:

- Collect wildcard receives
- Delay non-blocking sends that may match the receives
- Issue other non-interfering operations (such as the `Barrier`s)
- Finally, when forced (in the example in Figure 4, the forcing occurrs when all processes are at their `Wait` statements), ISP will perform the dynamic re-writings, followed by the two replays.

The example of Figure 4 illustrates many aspects of POE:

- *Delaying of operations:* `Irecv` was delayed in this example;
- *Out of order issue:* the `Barrier`s were issued out of program order;
- *Being forced to compute the full extent of non-determinism:* In our example, when all processes are at *fence* instructions – meaning, instructions such as `Wait` and `Barrier` – the POE algorithm knows that none of the sends coming *after* the fence can match the wildcard `Irecv`. (We will define a *fence* later on in Section 3.2.) This allows POE to compute the full extent of non-determinism on-the-fly.
- *Replay:* For each case of non-determinism, ISP only remembers the choices by keeping a *stack trail*. The full MPI runtime state cannot be remembered (it is too voluminous, spread over clusters, and consists of the entire state of a large machine). Thus, ISP always resorts to the approach of starting from the initial state (known) and computing up to a choice point, and then branching differently

```
P0                  P1                      P2
--                  --                      --
Isend(to:1,req);    Irecv(from:*,req);      Barrier;

Barrier;            Barrier;                Isend(to:1,req);

Wait(req);          Recv(from:2);           Wait(req);
```

Figure 5: POE Illustration

at the choice point. This approach has been employed in almost all *stateless dynamic verification approaches* that were pioneered in [1].

2 Illustration of POE through an Example

We will now explain at some length how POE will treat the example in Figure 5. This example will help understand the semi-formal description of POE given in Section 3.

- POE starts with `P0`.
- It will collect but does not issue `Isend` into the MPI runtime. Since this is a non-blocking call, POE stays with `P0`.
- It will collect but does not issue `Barrier` into the MPI runtime. POE now realizes that `P0` is at a *fence*, and hence it switches to `P1`. We define an instruction I to be a *fence* if all later instructions in program order are guaranteed to be matched only after I is matched. In our example subset of MPI, Barrier and Wait are fences. In general, all blocking MPI calls are fences. We prefer the term *fence* in lieu of *blocking call* because normally a `Barrier` is viewed as a *collective call* and not a blocking call in the MPI parlance.
- In the same manner, POE collects `Irecv` and `Barrier` from `P1`, then switches over to `P2`.
- It collects `Barrier` from `P2`.
- Now that all processes are at a fence, we form the currently formable match sets. There is only one match set – namely, the `Barrier` instructions. A *match set* is a collection of MPI commands that "go together," or are "performed together" – for instance, all matching barriers, a send and its receive, etc. There are also singleton match sets, for example a `Wait` whose Isend/Irecv have drained/filled their message buffers.

  Note that we *do not* form a match set out of `Isend` of `P0` and `Irecv` of `P1` because since this barrier is also "crooked," `Isend` of `P2` can match `Irecv` of `P1`. Wildcard receive match sets are formed *only* (i) when all processes are on fences, **and** (ii) a non wild-card match set cannot be formed. This fact will be addressed once again in Section 3.
- Since we could form a non wild-card match set (namely the match set of `Barriers`), we restart the POE algorithm, coming back to `P0`, and collecting (but not issuing) its `Wait`.
- Since `Wait` is a fence, POE shifts to `P1`'s `Recv`, which is like `Irecv` fused with an immediately following `Wait` (and hence this is also a fence).
- Then POE moves over to `P2` and continues working on `P2` (since the other processes are at fences. When `Wait` from `P2` is encountered, all processes are once again at fences.
- At this point, POE can form a set of match sets involving `Isend(to:1,req)`, `Irecv(from:*,req)`, and `Isend(to:1,req)`. The match set members are {`Isend(to:1,req)`, `Irecv(from:0,req)`}, and

{Isend(to:1,req), Irecv(from:2,req)} . This is a wildcard match set, and no other non wild-card match set exists. Hence we can now be sure that Irecv(from:*) of P1 will have no more matching sends! (If there were one more process in this program and through that we had a non wild-card match set, we could have executed that, moved forward, and perhaps found yet another send matching this wildcard receive.)

- POE backtracks over these match set members. If it proceeds via the second of these match sets, *i.e.* {Isend(to:1,req),Irecv(from:2,req)}, then Recv(from:2) no longer has a communicating partner, and this results in a deadlock.

  The reader is invited to consider the other match set which does not reveal a deadlock.

3 Semi-Formal Description of POE

In our discussions so far, we have made many bold claims such as "changing the order between Isend and Barrier does not matter." We also loosely alluded to notions such as operations "finishing." The formal semantics of ISP are described in our FM 2009 paper. This section attempts to summarize the main aspects of POE.

3.1 States of MPI Calls

Each MPI call goes through up to four states: *issued* (notated as $\triangleright$), *returned* ($\triangleleft$), *matched* ($\Diamond$), and *completed* ($\bullet$). Some calls (*e.g.*, Isend) go through all these states, while others (*e.g.*, Wait) does not have a *matched*, or $\Diamond$ state, and hence its *completed*, or $\bullet$ state may be taken to be the same as its *matched* state. Here are pertinent details of these events:

issued ($\triangleright$): The MPI function has been issued into the MPI runtime.

returned ($\triangleleft$): The MPI function has returned and the process that *issued* this function can continue executing. For a non-blocking call, this event occurrs immediately after the issue event.

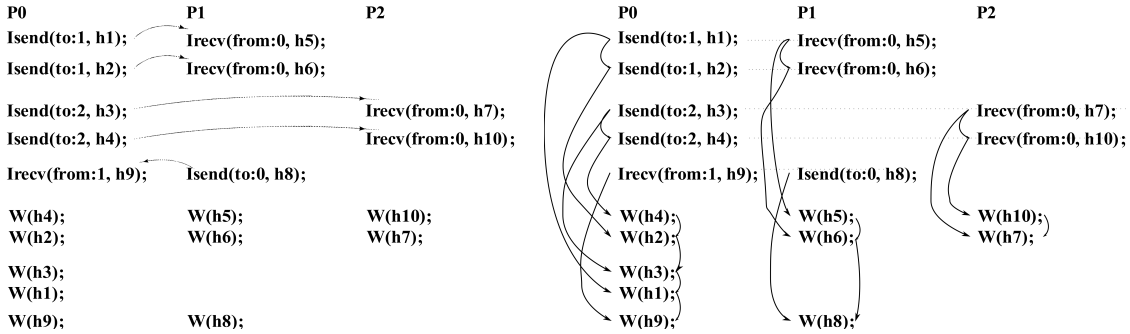
matched ($\Diamond$): Since most MPI functions usually work in a group (for example, a send from one process will be matched with a corresponding receive from another process), an MPI function is considered *matched* when the MPI runtime is able to match the various MPI functions into a group which we call a *match set*. Another example of a *match set* is the one contributed to by a Barrier (*B*): all the barrier calls match at some point. All the MPI calls in the *match set* will be considered as having attained the *matched* state.

complete ($\bullet$): An MPI function can be considered to be complete according to the MPI process that issues the MPI function when all visible memory effects have occurred (*e.g.*, in case the MPI runtime has sufficient buffering, we can consider an Isend to complete when it has copied out the memory buffer into the runtime bufer). The completion condition is different for different MPI functions (*e.g.*, the Irecv matching the Isend may not have seen the data yet, but still Isend can complete on the send side).

3.2 Happens-before

An MPI receive call "happens" when its match ($\Diamond$) event occurs. An MPI wait call "happens" when its completed ($\bullet$) event occurs. Without peering into these "inner events" associated with an MPI call, we can define an abstract *happens-before* relation between MPI calls. The details of such an *intra happens-before* relation for MPI was discovered by us (we call it simply by "MPI happens before"; in future papers, we will discuss MPI's inter happens-before relation). For simplicity, we consider only the non-blocking send (S), non-blocking receive (R), wait (W), and barrier (B).

- Two non-blocking sends (S) are happens-before ordered in program order if they target the same destination process.
- Two non-blocking receives (R) are happens-before ordered in program order if they source the same source process.
- A wildcard receive always happens-before before any following receive.
- A non-blocking send (S) and its wait (W) are happens-before ordered in program order.
- A non-blocking receive (R) and its wait (W) are happens-before ordered in program order.
- All instructions following (in program order) a barrier (B) are happens-before ordered after it.
- All instructions following (in program order) a wait (W) are happens-before ordered after it.
- Nothing else is happens-before ordered.



(a) Example Illustrating Happens-before

(b) The Happens-before Relation

Figure 6: Example Illustrating Happens-before

Consider the example in Figure 6(a). The horizontal dotted arrows depict the communications being attempted in this example. The full happens-before relation for this example is shown in Figure 6(b). There is a happens-before between `Isend(to:1,h1)` and `Isend(to:1,h2)` because they target the same process (namely `P1`), and likewise between `Isend(to:2,h3)` and `Isend(to:2,h4)` also. Each of these sends and their matching waits are happens-before ordered. However there is no happens-before between `Isend(to:1,h1)` / `Isend(to:1,h2)` and `Isend(to:2,h3)` / `Isend(to:2,h4)`. In an actual execution of this MPI program, it is possible for users to witness `Isend(to:2,h3)` or `Isend(to:2,h4)`. finish before `Isend(to:1,h1)` or `Isend(to:1,h2)` even start! This makes perfect sense from a performance point of view if the third and fourth send of `P1` are shipping far smaller messages.

Notice another surprising fact: there is no happens-before between the first four sends of `P0` and `Irecv(from:1,h9)`! Likewise, there is no happens-before between the first two receives of `P1` and the `Isend(to:0, h8)` of `P1`. This may be puzzling: users may wonder: what if `Isend(to:0,h8)` picks up a piece of data received by `Irecv(from:0,h6)` and tries to send it back to `P0`? Fortunately, this is not an issue because *it is illegal* for `Isend(to:0,h8)` to be using the data received by `Irecv(from:0,h6)`, because this non-blocking receive is still in progress, and its wait (namely `W(h6)`) hasn't yet been encountered! So really speaking, in this program, one may observe the `Isend(to:0,h8)` to `Irecv(from:1,h9)` communication as the first observable activity!

The POE maintains a dynamically evolving happens-before graph for each MPI process. Each time a new instruction is collected from an MPI process, POE extends the happens-before according to the rules described above. "Collect" means "advance the PC over the instruction." All happens-before edges are

intra process – we don't have the notion of an *inter happens-before* in POE (we have another algorithm under construction called POE_{MSE} that does maintain inter happens-before).

For two instructions X and Y, X is an ancestor of Y if X is happens-before ordered before Y; Y is called a descendent of X. Each instruction collected by POE may have been *matched* or may be as yet *unmatched*. Since happens-before is a partial order, at any point in time, POE can have a set of one or more instructions without ancestors. By the same token, it is also possible for a process to have more than one instruction with all its ancestors having been matched. Such instructions – those whose ancestors have all been matched – are *eligible to match*. When we say "pick an eligible-to-match instruction" in the following sequel, we mean one of those instructions.

Define an instruction I to be a *fence* if *all later* instructions in program order are guaranteed to be happens-before ordered after I (earlier on Page 5, we defined a fence in a much more informal manner). Call a process to be at a *fence* if its current instruction is a fence. For our purposes, a process is at a fence if it is at a blocking call, namely at a B or at a W. For simplicity, we consider only the following instruction types:

- non-blocking send (S)
- non-blocking receive - wildcard (R(*)) or non-wildcard (R). We abbreviate wildcard as WC.
- wait (W)
- barrier (B)

Let P_0,...,P_{N-1} be the N processes of an MPI program. Start with P_0 as the *current process*, and the first instruction of each process as the *current instruction* (instruction at the *current PC* whicl starts at 1). Since these instructions have no ancestors, they are also *eligible to match*. Finally, let state S (the argument of *POE* below) encapsulates the entire execution state of the MPI program including: the current PC and the current instruction of each process, the HB graph of each process, and the eligible to match set of each process. We express POE as a recursive algorithm, and in fact during the actual implementation, we do arrange for the recursive semantics to be achieved by backtracking and replaying. With these notions in place, here is the pseudo-code of POE.

3.3 Illustrations of POE

3.3.1 Simple Examples of "Auto Send"

As our first illustration, consider the following single process (rank) MPI pseudo-code program where, for simplicity, we have suppressed all message buffers:

```
P0: Irecv(from:0,h1); Barrier; Isend(to:0,h2); Wait(h2); Wait(h1);
```

Curiously, `P0` is trying to send to itself, which is allowed in MPI. The happens-before relation is as follows: from `Barrier` to all following instructions; from `Irecv` to its `Wait`, and from `Isend` to its `Wait`; from each `Wait` to all following instructions.

POE will collect `Irecv`, then collect `Barrier`. It forms the singleton non wild-card match set `Barrier` and issues it. This makes `Isend` to issue. Now a non wild-card match set of `Irecv` and `Isend` can be formed. These are issued. Now `Wait(h2)` becomes eligible to match. According to the MPI semantics, this `Wait(h2)` can return only when `Isend` has copied its buffer out of the process space. Where `Isend` puts this resulting data depends on the MPI implementation. If the MPI runtime has sufficient buffering of its own, then clearly this `Isend` can void its buffer. But since `Irecv` has been posted,

```
1:  POE(S) {
2:      if (any local assertion is violated in S)
3:          Error report; Return;
4:      if (all procs are at a finalize state in S)
5:          Check for MPI object leaks; Return;
6:      if (there is a proc not at a fence instruction in S)
7:          Let S' be obtained by advancing the PC of this proc, and extending the HB graph;
8:          POE(S');
9:      if (a non wild-card match set can be formed)
10:         Let S' be obtained by issuing the match set operations, and updating eligible to match;
11:         POE(S');
12:     // All procs are at fences and no non-WC match set is formable
13:     for (all WC match sets m)
14:         for (all dynamic rewrites of m resulting in a match set ms)
15:             Let S' be the state resulting from expanding this WC match set as follows:
16:                 Issue the constituent instructions of ms;
17:                 Update the Eligible to Match relation;
18:         POE(S');
19: }
```

Figure 7: POE Pseudo-code

it is required that `P0` itself pick up the data sent out by `Isend` (see **Note** below), allowing `Isend` (and its wait) to return. This program executes deadlock-free in all MPI implementations.

> **Note:** *We are unsure whether the standard requires this.* In the ISP implementation, we package a fake `Wait` along with non-blocking operations because we have encountered at least one MPI implementation where `Isend` and `Irecv` do not progress till their waits are seen. In this case, ISP nullifies the real `Wait` when it comes along later. If such a `Wait` does not come along, the faked wait still does not count as far as resource leak reporting goes (*i.e.*, we may have faked a wait, but if a real wait does not come along, it is still flagged as a resource leak).

Let us discuss further variations of this program, all discussed with respect to MPI runtimes that do not provide sufficient run-time buffering. A deadlock-free variant of the above program is

```
P0: Irecv(from:0,h1); Barrier; Isend(to:0,h2); Wait(h1); Wait(h2);
```

Here, waiting for `Irecv` first through `Wait(h1)` is safe, because eventually this `Irecv` is supposed to finish (again, subject to **Note** above).

However, the following version deadlocks, because the `Wait` of the receive has to finish before `Isend` can even be initiated.

```
P0: Irecv(from:0,h1); Wait(h1); Barrier; Isend(to:0,h2); Wait(h2);
```

3.3.2 Example in Figure 6

From the HB graph in Figure 6, it is easy to see that all of the following can be issued into the MPI runtime even in the very first step of execution:

- `Isend(to:1, h1)`
- `Isend(to:2, h3)`
- `Isend(to:0, h8)`
- `Irecv(from:0, h5)`
- `Irecv(from:1, h9)`
- `Irecv(from:0, h7)`

In particular, as pointed out earlier, one may observe the `Isend(to:0,h8)` to `Irecv(from:1,h9)` communication as the very first activity in this MPI program.

3.3.3 Cross-Coupled Processes

This section illustrates one very non-obvious aspect of MPI programs, and why the recursive invocation beginning at Line 14 of Figure 7 is crucial for handling such examples. Here is the point. At line 13, we may have two distinct wild-card match sets m1 and m2. In particular, m1 itself may be comprised of one wildcard receive and two matching sends, and likewise m2 may be comprised of one (seemingly completely unrelated) wildcard receive and its own two matching sends. At line 14, we go with m1, calculate S$'$ using it, and issue POE(S$'$) recursively at line 18. Does m2 remain "frozen" at its current state – i.e. are we really sure that the set of sends that match the wildcard receive within m2 remains at the initial two we discovered while at line 13? The answer is no! When we go with m1, calculate S$'$ using it, and issue POE(S$'$) recursively at line 18, we may engage in new code that issues *yet another send* (call it "new send") that also matches the wildcard receive within m2! Since POE has no such 'look-ahead knowledge,' it has to play conservatively, doing what we express in Figure 7. That is, when we go with m1, calculate S$'$ using it, and issue POE(S$'$) recursively at line 18, we are starting afresh at line 1, and this will result in the "m2 match set" being re-calculated in the light of the new send that we would have discovered.

All this is brought to clearer light by the following example. We employ an abstract syntax showing the code of each MPI process as a straightline program (Figure 8), suppressing all intervening C statements as before. Each MPI call is decorated with two subscripts, namely the process index and the rank within a process. In this example, the first send of P_1, *i.e.* $S_{1,1}(0)$, aims to send a data payload (not shown) to process P_0 (the argument 0 of $S_{1,1}$ signifies this). Its corresponding wait is $W_{1,2}(\langle 1,1\rangle)$ (we often employ pairs such as $\langle 1,1\rangle$ to denote handles). We leave out the "to" and "from" designators from sends and receives.

P_0	P_1	P_2	P_3
$R_{0,1}(*)$	$S_{1,1}(0)$	$R_{2,1}(*)$	$S_{3,1}(2)$
$W_{0,2}(\langle 0,1\rangle)$	$W_{1,2}(\langle 1,1\rangle)$	$W_{2,2}(\langle 2,1\rangle)$	$W_{3,2}(\langle 3,1\rangle)$
$R_{0,3}(*)$	$S_{1,3}(2)$	$R_{2,3}(*)$	$S_{3,3}(0)$
$W_{0,4}(\langle 0,3\rangle)$	$W_{1,4}(\langle 1,3\rangle)$	$W_{2,4}(\langle 2,3\rangle)$	$W_{3,4}(\langle 3,3\rangle)$

Figure 8: Example Requiring Look-ahead Ample Sets

In Figure 8, the match of $R_{0,1}(*)$ is initially $S_{1,1}(1)$, and that of $R_{2,1}(*)$ is $S_{3,1}(2)$. Pursuing the first match set eventually makes $S_{1,3}(2)$ also eligible for a match with $R_{2,1}(*)$. There is a similar expansion of the match set of $R_{0,1}(*)$ possible from $S_{3,3}(0)$ when the match between $R_{2,1}(*)$ and $S_{3,1}(2)$ is pursued. Since POE does not have such look-ahead knowledge, we will handle things conservatively on Line 13.

While correct (we will formalize the correctness proof elsewhere), what performance impact does this decision have? Observe that we can end up doing 'part of the work' at first, and then the 'whole of the work' properly again. More specifically,

- We will match $R_{2,1}(*)$ and $S_{3,1}(2)$ initially, and then pursue $R_{0,1}(*)$ along with $S_{1,1}(1)$ *and* $S_{3,3}(0)$.
- Then we will match $R_{0,1}(*)$ and $S_{1,1}(0)$ initially, and then pursue $R_{2,1}(*)$ along with $S_{3,1}(2)$ *and* $S_{1,3}(2)$.

In other words, while the match set consisting of $R_{2,1}(*)$ and $S_{3,1}(2)$, and that consisting of $R_{0,1}(*)$ and $S_{1,1}(0)$ appear completely isolated, we recursively expand out all transitions emanating from the initial state, thus foregoing the advantages of partial order reduction. The advantage given up is of course necessary if/when we encounter $S_{1,3}(2)$ and $S_{3,3}(0)$ that induce further dependencies. However, in the absence of "cross coupled" dependencies such as $S_{1,3}(2)$ and $S_{3,3}(0)$, we might have pursued (say) $R_{2,1}(*)$ and $S_{3,1}(2)$, and from the resulting state pursue $R_{0,1}(*)$ and $S_{1,1}(0)$. Such an improved algorithm, namely POE_{MSE}, has been conceived and prototyped, but is not part of the released version of ISP.

There are many practical examples where the initial set of wildcard match sets are truly independent. For example, the MADRE (Memory Aware Data Redistribution Engine of Siegel) and the MPI-BLAST (genome sequencing using MPI) code bases both exhibit this pattern. In both these applications, there is absolutely no 'cross-coupling' of dependencies whatsoever. Naturally, when we apply POE, we obtain unnecessary interleaving explosion. This means that an algorithm such as POE_{MSE} will be important in our continued use of ISP.

3.3.4 ISP: Additional Practical Examples and Concluding Remarks

Several examples will be presented during the course of our tutorial. These examples are all well described in our URL cited on our first page. The user manual of ISP describes the usage of ISP using interactive commands.

4 Shared Memory Thread Program Verification using Inspect

This section briefly describes one example that will be examined in some detail during our EuroPVM/MPI tutorial. Consider Figure 9 in which two threads attempt to acquire two different locks, namely `lock` and `mutex`. This program will be instrumented by Inspect through its front end that is based on the Berkeley CIL framework. This front end automatically detects and wraps global variables as well as all Pthread calls. Inspect employs the classic dynamic partial order reduction algorithm [3] with several efficiency enhancements. The deadlock detected involves the scenario where thread A is holding `lock` whereas thread B is about to attempt to acquire `lock` – but after having acquired `mutex`. Alas, thread A cannot release `lock` unless it can acquire `mutex` – causing a circular wait.

```
void * thread_A(void* arg)
{
  pthread_mutex_lock(&mutex);
  A_count++;
  if (A_count == 1)
    pthread_mutex_lock(&lock);
  pthread_mutex_unlock(&mutex);

  pthread_mutex_lock(&mutex);
  A_count--;
  if (A_count == 0)
    pthread_mutex_unlock(&lock);
  pthread_mutex_unlock(&mutex);
}
```

```
void * thread_B(void* arg)
{
  pthread_mutex_lock(&mutex);
  B_count++;
  if (B_count == 1)
    pthread_mutex_lock(&lock);
  pthread_mutex_unlock(&mutex);

  pthread_mutex_lock(&mutex);
  B_count--;
  if (B_count == 0)
    pthread_mutex_unlock(&lock);
  pthread_mutex_unlock(&mutex);
}
```

Figure 9: Example Pthread program with a deadlock

4.1 A Inspect User Manual

`Inspect` is a tool for systematic testing multithreaded C programs under specific inputs. It can systematically explore different interleavings of multithreaded programs under specific inputs by repeatedly executing the program. `Inspect` is aimed to reveal concurrency related bugs, such as data races, deadlocks, etc. (Because of the lack of a C++ source code transformer, Inspect cannot automatically instrument C++ programs. With manual instrumentation, we can still use Inspect to check multithreaded C++ programs. But you need to know more about how Inspect works to do the manual instrumentation.)

4.1.1 Download and Installation

The file `./inspect-0.1.tar.gz`inspect-{*version*}.tar.gz contains the complete source distribution. Inspect is written in C++ and Ocaml. You need `http://caml.inria.fr`Ocaml release 3.10 or higher, and a C++ compiler to build Inspect. Inpect has been tested under Linux systems. It has not been tested under Windows or other Unix systems yet. The source can be compiled with the following commands:

```
tar xzf inspect-{version}.tar.gz
cd inspect-{version}
make distclean
make
```

4.2 How to Use Inspect

The following command line shows how to use Inspect. First we instrument the multithreaded code. Then we compile the instrumented code, and link with a wrapper library. After this, we have an executable which can be systematically tested using inspect.

```
bin/instruemnt   examples/dpor-example1.c
bin/compile      dpor-example1.instr.c
./inspect ./target
```

4.3 Emacs Interface

The Emacs interface is found in the file `inspect-mode.el`. The file `emacs-config`, generated by `make`, contains a minimum emacs configuration whose contents should be added to the user's emacs startup script.

Customization variables:

- `inspect-path` is the absolute path the the Inspect directory.
- `inspect-instr-user-args` is a list of string arguments to pass to the instrumentation.
- `inspect-run-user-args` is a list of string arguments to pass to Inspect.

To run Inspect, select the a program source buffer and invoke the command (M-x) `inspect-run`. The buffer `*inspect-run*` will display progress information including the current test run and the number of races and deadlocks currently found. Inspect may be terminated at any time by pressing `k` in the `*inspect-run*` buffer.

The *inspect-run* buffer displays the traces corresponding to the first run (*inspect-first-run*) and all races (*inspect-*run*-race-*count**), deadlocks (*inspect-*run*-deadlock*), and assertion failures (*inspect-*run*-assert*). It also includes the output of the program (*inspect-*run*-output*) if there is any.

Pressing enter or clicking the left mouse button will display the selected buffer. Pressing backspace will return to the main buffer *inspect-run*. Alternatively, buffers may be selected using C-x B or C-x C-b. The source line of a trace entry can be displayed by pressing enter or clicking the left mouse button.

Debugging buffers:

- *inspect-compile-debug* contains the command line for the instrumentation and its output.
- *inspect-run-debug* contains the command line for the invocation of Inspect and the uninterpreted output.

4.3.1 Related Projects

- http://cm.bell-labs.com/who/god/verisoft/ Verisoft
- http://research.microsoft.com/chess CHESS

Acknowledgements: Thanks to Robert M. Kirby, Sarvani Vakkalanka, Yu Yang, Anh Vo, Michael DeLisi, Sriram Aananthakrishnan, Alan Humphrey, Chris Derrick, Simone Atzeni, Grzegorz Szubzda, Subodh Sharma, and Geof Sawaya.

Funding Acknowledgements: This research was supported by Microsoft, NSF "CSR-SMA: Toward Reliable and Efficient Message Passing Software Through Formal Analysis," "CPA-DA: Formal Methods for Multi-core Shared Memory Protocol Design," and "MCDA: Formal Analysis of Multicore Communication APIs and Applications," and by SRC Task ID TJ 1847.001 "Formal Specification, Verification, and Test Generation for Multi-core CPUs," August 2008, and and Task ID TJ 1993, "MCDA: Formal Analysis of Multicore Communication APIs and Applications."

References

[1] Patrice Godefroid. Model Checking for Programming Languages using Verisoft. POPL, 1997, 174-186.

[2] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.

[3] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.

[4] The Java Pathfinder. http://javapathfinder.sourceforge.net

[5] Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Pages 446–455, 2007.

[6] CHESS: Find and Reproduce Heisenbugs in Concurrent Programs. http://research.microsoft.com/en-us/projects/chess/

[7] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Chao Wang. Automatic Discovery of Transition Symmetry in Multithreaded Programs using Dynamic Analysis. SPIN 2009, Grenoble, June 2009.

[8] MPI Standard 2.1. http://www.mpi-forum.org/docs/.

[9] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. *CAV 2008.*

[10] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. *PPoPP 2009.*

[11] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, and R. M. Kirby. Scheduling considerations for building dynamic verification tools for MPI. *PADTAD-VI 2008.*

[12] Sarvani Vakkalanka et al. Static-analysis Assisted Dynamic Verification of MPI Waitany Programs (Poster Abstract). Accepted in EuroPVM/MPI, September 2009.

[13] Anh Vo et al. Sound and Efficient Dynamic Verification of MPI Programs with Probe Non-Determinism. Accepted in EuroPVM/MPI, September 2009.

[14] R. Palmer, M. DeLisi, G. Gopalakrishnan, R. M. Kirby. An Approach to Formalization and Analysis of Message Passing Libraries *FMICS 2008, 164–181*

[15] Guodong Li et al. Formal Specification of the MPI 2.0 Standard in TLA+. Under Submission. http://www.cs.utah.edu/formal_verification/mpitla/.

[16] Leslie Lamport. Specifying Systems: The TLA Language and Tools. Addison-Wesley, 2004.

[17] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *SuperComputing (SC)*, 1996.

[18] S. F. Siegel and A. R. Siegel. MADRE: The Memory-Aware Data Redistribution Engine. *EuroPVM/MPI 2008.*

[19] Workshop on Exploiting Concurrency Efficiently and Correctly, Grenoble, June 2009. Discussion of Challenge Problems. http://www.cs.utah.edu/ec2.

[20] P. Pacheco Parallel Programming with MPI *Morgan Kaufmann, 1996, ISBN 1-55860-339-5*

[21] Michael DeLisi. Umpire Test Suite Results using ISP. http://www.cs.utah.edu/formal_verification/ISP_Tests/

[22] Leslie Lamport. Time Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM. 21(7):558-564, 1978.

[23] S. F. Siegel. Efficient Verification of Halting Properties for MPI Programs with Wildcard Receives. *VMCAI 2006, 413–429*

[24] P. Georgelin, L. Pierre, and T. Nguyen. A Formal Specification of the MPI Primitives and Communication Mechanisms. Rapport de Recherche LIM 1999-337, Marseille, Oct 1999.

[25] http://www.cs.utah.edu/formal_verification/ISP-release/.

[26] mpiBLAST: Open-Source Parallel BLAST. http://www.mpiblast.org/.

[27] The IRS Benchmark Code. https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/.

[28] R. Martin and A. Manohar. Slack Elasticity in Concurrent Computing *n Intl Conf. on Mathematics of Program Construction. LNCS 1422*

[29] Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. MCC - A runtime verification tool for MCAPI user applications. Accepted in FMCAD 2009, Austin.

[30] The Multicore Communications API (MCAPI). http://www.multicore-association.org