Scheduling Considerations for Building Dynamic Verification Tools for MPI *

Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan and Robert M. Kirby
School of Computing, University of Utah, Salt Lake City, UT 84112
{sarvani, delisi, ganesh, kirby}@cs.utah.edu

ABSTRACT

Dynamic verification methods are the natural choice for formally verifying real world programs when model extraction and maintenance are expensive. Message passing programs written using the MPI library often fall under this category, especially when these programs have to be verified after being ported to new platforms and iteratively optimized. However, implementing dynamic verification tools for MPI requires solving many problems pertaining to the scheduling realities of the MPI runtime. In this paper, we describe the progression of our ideas during the development of our dynamic verification tool for MPI programs, called *in-situ partial order (ISP)*. Each idea developed in this progression relates to our dual goals of ensuring full coverage of all *representative interleavings* (in the sense of partial order reduction) among MPI processes, and forcing the MPI runtime to *affect* these interleavings. We briefly examine similar issued faced by other builders of dynamic verification tools. We conclude with observations backing the growing importance of addressing scheduling issues in future dynamic verification tools for various concurrency APIs.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model Checking; D.1.3 [**Concurrent Programming**]: Distributed programming

General Terms

Verification, Distributed Programming, Message Passing

Keywords

Dynamic Verification, MPI, Partial Order Reduction, Model Checking

*Supported in part by NSF CNS-00509379, and Microsoft HPC Institutes Program.

PADTAD'08, July 20–21, 2008, Seattle, Washington, USA.

1. INTRODUCTION

The growing popularity of dynamic formal verification tools (*e.g.*, [1, 2, 3, 4, 5, 6, 7]) is indicative of the relative merits of verifying concurrent software at the level of the final source code. Most real world concurrent software is written using complex APIs such as MPI, OpenMP, or PThreads. Capturing the semantics of these API calls in modeling languages such as Promela is both tedious and error-prone. Capturing the subtle semantics of the C constructs that "weave" these API calls together is equally hard. Instead, under dynamic formal verification, one gets to run the software with very little modeling effort on the part of the user. One also avoids the overhead of model maintenance after every performance tuning or porting step. Furthermore, one may discover incidental bugs in C code that surrounds the API calls; these bugs may be masked under many abstraction methods that convert C to formal modeling notations before verification.

In addition to the above advantages, dynamic verification can facilitate search optimizations such as partial order reduction to be conducted more effectively (*e.g.*, [3, 5, 8]). For instance, if two processes P1 and P2 have, as their next executable actions, the statements `a[i]++` and `a[j]--`, the values of `i` and `j` (which govern the commutability of these actions) are often known only at runtime. Likewise, in the context of the MPI library, if there are three MPI send operations that potentially target an MPI wildcard receive statement, whether they actually do or not target that statement is a complex function of (i) the actual computed send targets (many MPI programs compute their send targets), (ii) the manner in which conditionals resolve, and (iii) the dynamic out of order semantics of MPI [8].

This paper is on the design of a dynamic verification tool for MPI that affects partial order reduction, taking advantage of the runtime information. Our focus on MPI is justified by its growing popularity: it is the dominant programming model for large-scale parallel machines. In our work, we ran into many problems relating to controlling the MPI runtime. In general, the problem of controlling the runtime of the concurrency API being used varies in degrees of difficulty, depending on the API at hand. For dynamic verification systems built around Java, the problem often turns into building backtrackable virtual machines and, as we surmise from the general lack of any special mention, perhaps is 'surprise-free' for the most part. This approach is not entirely feasible for systems built around languages such as C. To handle such languages, one practical approach is (i) to locate the global interaction points of each process,

(ii) put wrappers so that a scheduler can intercept any execution of code at these interaction points, and (iii) let the scheduler orchestrate one interleaved execution, and later, as many variant interleavings as desired. This approach is followed, for instance, in [3, 5]. In [5], we document additional details of such an approach, including: (i) making sure that `malloc`s and other similar operations return the *same* addresses for heap objects across various interleavings and (ii) dealing with open inputs. We note that dealing with open inputs remains an important problem; however, this problem is not unique to dynamic verification. The other precautions – *e.g.* ensuring deterministic replays including for `malloc`s – do need careful handling, but are not related to the scheduling details of the runtime, which is the subject of this paper.

Scheduling Realities: The scheduling realities of a dynamic verification tool are a function of the features (sometimes more aptly called peculiarities) of the API in question. The authors of Chess [9] briefly mentioned the steps they had to take to ensure that the Win32 API that they were orchestrating as part of their dynamic verification system behaved as desired. In particular, the Chess scheduler has to have a means to determine whether a certain Win32 API action would be scheduled by the runtime if invoked from the outside (without such a mechanism, one could deadlock the dynamic verification tool by issuing a blocking API operation that will not be scheduled for execution). This is just about the only interesting scheduling-related detail we can narrate with respect to others' work. We speculate that a sufficient number of people are yet to face these difficulties, as they are not yet working on APIs with "interesting semantics."

Roadmap: In this paper, we show that the scheduling details of a dynamic verification system for MPI can be extensive. We first provide a summary of the *current* features of our tool *in-situ* partial order (ISP) [8], to provide some context for the rest of this paper (Section 2). We point out that ISP has become practically useful, only thanks to our being able to solve a number of problems pertaining to the schedule orchestration of MPI.

We then wind back to the early days of ISP and present the progression of our ideas leading to our present tool version. We begin with what we had to do to 'tame' the progress engine of MPI in dealing with MPI's one-sided communication (Section 3). We then describe the steps taken by our first version of ISP in having an implicit deadlock detection method (Section 4), and present some details of the POE[1] algorithm used in ISP. This is followed by the steps necessary to implement `MPI_WAITANY` and `MPI_TESTANY` (Section 5), and the problems caused by uneven sizes of data (Section 6). We describe how a Visual Studio user interface of ISP was impacted by the POE algorithm from the point of view of presenting meaningful error traces during dynamic verification (Section 7). We conclude (Section 8) by summarizing the tricks anticipated to be needed in the context of a runtime verifier for OpenMP and MPI with multiple threads.

2. ISP OVERVIEW

We now provide a summary of recent results obtained using ISP. As shown by experiments on our website [10], of the 69 Umpire tests,[2] 30 contain deadlocks, and ISP detects all of them, while exploring a very small number of interleavings. In contrast, conventional testing-based tools such as Marmot [11] have been observed to miss deadlocks in eight of these tests, despite being run multiple times. When these tests were run under MPICH 2.0 repeatedly, the deadlock detection success was unpredictable. Despite great advances in noise-maker design [12], tools that rely on perturbing schedules cannot guarantee coverage.

```
P0:  MPI_Isend(to P1, data = 22); ...rest of P0...
P1:  MPI_Irecv(*, x); if (x==22)
                      then error1
                      else ...rest of P1...;
P2:  MPI_Isend(to P1, data = 33); ...rest of P2...
```

Figure 1: Simple MPI Example Illustrating Wildcard Receives

In other recent experiments, we have run ISP on the Monte Carlo calculation of Pi [13], and the Game of Life example used in the EuroPVM/MPI 2007 Tutorial [14]. The Game of Life code is a 500+ line MPI program. It was handled by ISP without any manual modeling effort whatsoever. ISP examines exactly one interleaving for this example, thanks to its aggressive partial order reduction algorithm and the fact that these examples do not employ wildcard receives. (ISP's partial order reduction algorithm is a new algorithm called POE – partial order avoiding elusive interleavings – and is the subject of [8].) We had instrumented ISP to detect resource leaks, and ISP's search guarantees the absence of resource leaks in this example.

PMPI based scheduling: We use the well-known PMPI mechanism, often used for performance studies, to support runtime model checking in ISP. We introduce an extra process called the *verification scheduler*. ISP provides its own version of "MPI_f" for each MPI function f. Within MPI_f, we arrange for handshakes with the scheduler that realizes the POE algorithm. When the scheduler finally gives permission to fire f, we invoke PMPI_f from within our version of MPI_f. The MPI runtime only sees the PMPI_f calls.

2.1 Examples Illustrating ISP

2.1.1 Wildcard Receives

The short MPI program pseudo-code in Figure 1 helps illustrate some aspects of ISP. In this example, MPI processes `P0` and `P2` are targeting `P1` which entertains a 'wildcard match,' *i.e.*, can receive from *any* process that has a concurrently enabled `MPI_Isend` targeting `P1`. As soon as one such send is chosen (say `P0`'s), the other send is not eligible to match with this receive of `P1` (it has to match another receive of `P1` coming later). This disabling behavior of the sends induces a dependency between them, as can be seen from the fact that the particular send that matches may or may not cause `error1` to be triggered. Consider some $i < j < k$, and a trace t where the ith action of t, namely t_i, is P2's send, and similarly t_j is P1's receive, and t_k is P0's send. In this trace, it is not necessary that P1's receive is

[1]Partial order avoiding elusive interleavings.

[2]While the program sizes run through ISP so far are a far cry from real world MPI program sizes, ISP is the first dynamic verification tool for MPI that has a sound partial order reduction algorithm. We expect to be able to handle many 'real world' MPI programs within the next few months.

matched with P2's send just because t_i is executed before t_k. MPI implements its own buffering mechanism that can cause one send to race ahead of the other send. Formally, unlike in other works (*e.g.*, [3]), *the program order* of MPI calls does not imply *happens-before* [15] of the MPI call executions. (The per-process ordering of MPI operations is the intra completes-before or **IntraCB** ordering described in Section 2.2.) Hence, it is possible that t_j is matched with t_k. There is no way in an MPI run-time (short of making intrusive modifications to the MPI library, which is often impossible because of the proprietary nature of the libraries) to force a match either way (both sends matching the receive in turn) by just changing the order of executing sends from P2 and P0. This illustrates one reality we faced and solved in the development of ISP. More details are presented in Section 5. We now present some aspects of MPI barriers and how related issues were handled by ISP.

2.1.2 MPI Barriers

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. 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 the execution. While these rules match the rules followed by other languages and libraries in supporting their barrier operations, in case of MPI, it is possible for a process P_i to have an operation OP before its barrier call, for another process P_j to have an operation OP' after P_j's matching barrier call, and where OP can observe OP's execution. This means that OP can, in effect, complete after P_i's barrier has been invoked. This shows that the program ordering from an operation to a following barrier operation need not be obeyed during execution. This is allowed in MPI (to ensure higher performance), as shown by the example in Figure 2, and requires special considerations in the design of POE. In this example, one `MPI_Isend` issued by `P0`, shown as `S0`, and another issued by `P2`, shown as `S2`, target a wildcard receive issued by `P1`[3]. The following execution is possible: (i) `S0(to P1, h0)` is issued, (ii) R(*, h1) is issued, (iii) each process *fully* executes its own barrier, (`B0`, `B1`, or `B2`), and this "collective operation" finishes (all the `B`'s indeed form an atomic set of events), (iv) `S2(to P1, h2)` is issued, (v) now both sends and the receive are alive, and hence `S0` and `S2` become dependent, requiring a dynamic algorithm to pursue both matches. Notice that `S0` can finish after `B0` and `R` can finish after `B1`.

Completes Before Ordering: This relation has two aspects: intra completes-before (IntraCB), and inter completes-before (InterCB). To simplify things a bit, the **IntraCB** ordering captures the fact that MPI respects program ordering between any MPI operation x belonging to the set $\{\text{barrier}, \text{wait}, \text{test}\}$ and the MPI operation immediately following x in program order. (The **InterCB** ordering is described in Section 2.2.) A dynamic verification algorithm for MPI must therefore maintain a *completes-before* relation, and use it to determine, at runtime, all senders that can match a wildcard receive (we briefly describe the completes-before relation in Section 2.2).

[3]While not central to our current example, we also take the opportunity to illustrate how the handles `h0` through `h2`, and `MPI_Wait` (`W`) are used.

```
P0:  S0(to P1, h0) ; B0 ;                W(h0) ;
P1:  R (*, h1)     ; B1 ;                W(h1) ;
P2:                  B2 ; S2(to P1, h2); W(h2) ;
```

Figure 2: Illustration of Barrier Semantics and the POE Algorithm

2.1.3 POE Algorithm Overview

We now present an overview of the POE algorithm, as implemented by our verification scheduler (called the POE scheduler) that can intercept MPI calls and send them into the MPI run-time as and when needed:

- The POE scheduler executes C program statements along each process. All C statements are executed in program order. When the scheduler encounters an MPI operation, it simply records this operation, but does not execute it. This process continues until the scheduler arrives, within each process, at an MPI operation that is program ordered with respect to some previously collected (but not issued) MPI operation (we call these points *fences*).
- While at a fence point for all processes, since all senders that match a wildcard receive are known, *rewrite* the receives into specific receives. In our example, `R(*)` is rewritten into `R(from P0)` and `R(from P2)`.
- Form *match-sets*. Each match-set is either a single big-step move (as in operational semantics) or a set of big-step moves. Each big-step move is a set of actions that are issued collectively into the MPI run-time by the POE scheduler (we enclose them in $\langle\langle \ldots \rangle\rangle$). In our example, the match-sets are:
  - $\{$ $\langle\langle$ `S0(to P1)`, `R(from P0)` $\rangle\rangle$, $\langle\langle$ `S2(to P1)`, `R(from P2)` $\rangle\rangle$ $\}$
  - $\langle\langle$ `B0`, `B1`, `B2` $\rangle\rangle$
- Execute the match-sets in priority order, with all big-step moves executed first. The execution of a big-step move consists of executing all its constituent MPI operations. When no more big-step moves are left, then for each remaining set of big-step moves, recursively explore (according to depth-first search) all the big-step moves contained in it. In our example, this results in the big-step move $\langle\langle$ `B0`, `B1`, `B2` $\rangle\rangle$ from being performed first. Subsequently, both the big-step moves in the set

  $\{$ $\langle\langle$ `S0(to P1)`, `R(from P0)` $\rangle\rangle$, $\langle\langle$ `S2(to P1)`, `R(from P2)` $\rangle\rangle$ $\}$

  are pursued.

Thus, one can notice that POE never actually issues into the MPI run-time any wildcard receive operations it encounters. It always dynamically rewrites these operations into receives with specific sources, and pursues each specific receive paired with the corresponding matching send as a match-set in a depth-first manner.

```
MPI_Generic_Func(arg1, arg2...argN) {
  sendToSocket(pID, Generic_Func, arg1,...,argN);
  while(recvFromSocket(pID) != go-ahead)
    MPI_Iprobe(MPI_ANY_SOURCE, 0, MPI_COMM_WORLD...);
  return PMPI_Generic_Func(arg1, arg2...argN);
}
```

Figure 3: PMPI instrumentation pseudocode

2.2 IntraCB and InterCB Relations of MPI

The **IntraCB** and **InterCB** orderings of MPI are best introduced with the help of the following example.

```
P0: MPI_Irecv(*, x, &h);  Wait(&h); Barrier;
P1: MPI_Isend(to P0, 33); Barrier;  ...rest of P1...
P2: ...some code...       Barrier;  MPI_Isend(to P0, 22);
```

In this example, there is an IntraCB edge from `Wait` to `Barrier`. Hence, `Barrier` cannot be crossed until the `MPI_Irecv` finishes. Therefore the `MPI_Isend` from `P2` cannot issue. Therefore, `MPI_Irecv` has to finish based on the `MPI_Isend` from `P1` alone.

InterCB: The reasoning employed in this example highlights the need for the notion of *InterCB* edges. Here, the `MPI_Isend` of `P2` "wishes to match" the `MPI_Irecv` of `P0`. The only thing that prevents this is that the barrier orders `MPI_Irecv` to be before it, and `MPI_Isend` to be after it. This is the ordering defined by InterCB. Basically, InterCB builds on the IntraCB order through the `Barrier` statements. We mention InterCB for the sake of completeness, because many interesting analyses can be set up based on it. For instance, recently we have developed an algorithm (under submission) to detect whether barriers in MPI programs are functionally redundant or not.

3. ONE-SIDED COMMUNICATION

Our work on dynamic verification methods leading to the present ISP began with the work in [16] where we presented our approach to verifying MPI programs with one-sided (weak shared memory) communication. This was our first attempt at dealing with scheduling and forward progress during dynamic verification of MPI.

The pseudocode in Figure 3 captures how the MPI call of a process (generically called `Generic_Func`) is processed. Note that the MPI function call is intercepted by the profiling library. It then conveys the process id (`pID`), the call type (`Generic_Func`), and the remaining arguments to the scheduler through the `sendToSocket` call. By way of reply, the scheduler either provides a 'go-ahead' or a 'loop' to the appealing processes. A 'loop' signal indicates that the appealing process must make an `MPI_Iprobe` call. The `MPI_Iprobe` call is just one side-effect free mechanism that causes control to enter the MPI progress engine to process all queued-up events within it, ensuring forward progress. `MPI_Iprobe` is needed with MPICH2 in order to cause progress to occur on communication with other processes, because MPICH2 does not use an asynchronous progress thread in its progress engine.[4]

4. IMPLICIT DEADLOCK DETECTION

[4]`MPI_Iprobe` does not have a version corresponding to `MPI_Generic_Func`; otherwise, it would cause an infinite loop when these `MPI_Iprobe`s are trapped.

```
0 : // * means MPI_ANY_SOURCE
1 : if (rank == 0)
2 : { MPI_Irecv(buffer, *, &req);
3 :   MPI_Recv (from 2);
4 :   MPI_Wait (&req); }
5 : else if (rank == 1)
6 : { MPI_Isend(buffer1, to 0, &req);
7 :   MPI_Wait (&req); }
8 : else if (rank == 2)
9 : { MPI_Isend(buffer2, to 0, &req);
10:   MPI_Wait (&req); }
```

Figure 4: Example for Implicit Deadlocks

```
2 : MPI_Irecv(buffer, *, &req);
6 : MPI_Isend(buffer1, to 0, &req);
9 : MPI_Isend(buffer2, to 0, &req);
3 : MPI_Recv (from 2);
7 : MPI_Wait (&req);
10: MPI_Wait (&req);
4 : MPI_Wait (&req);
```

Figure 5: MPI trace for Figure 4

As explained in Section 2.1.3, we avoided the inability to force certain communication matches through dynamic rewriting. Before this idea was developed, our approach to verify MPI programs through dynamic verification took the approach of *implicit deadlock detection*: (i) following [3], we implemented a DPOR algorithm for MPI, (ii) upon generating one interleaving based on a set of communication matches, we *predicted* what other interleavings/matches were possible, and predicted deadlocks that could occur during them. This approach was abandoned because it ignored some practical considerations – mainly that without actually executing those code paths, one would miss bugs introduced by the C statements (*e.g.*, a segmentation fault). However, it is worth recording the scheduling-related ideas in this approach.

For MPI programs that do not have any conditional dependence on the data received, it is possible to detect deadlock using just a single interleaving (trace). Consider the MPI program in Figure 4. This program has no dependence on the data received. An implicit deadlock mechanism runs the MPI program and generates a single trace of the MPI run as shown in Figure 5.

Once the trace is available, the IntraCB and InterCB edges can be used to detect the *co-enabled* sends and receives. The implicit detection algorithm works by recursively generating different interleavings *without* restarting the MPI program by generating the different possible match-sets using the co-enabled MPI sends and receives detected and hence generating possible interleavings.

The algorithm constructs the graph structure by adding the IntraCB and InterCB edges in parallel with the trace generation. Since the algorithm does an in-order issue (POE in contrast is an out-of-order issue algorithm), it is not possible to do any "source" re-writes for wildcard receives. If the MPI program deadlocks while generating the trace, the algorithm must use a timeout mechanism when it does not get any MPI operations from the processes. The deadlock detection happens on the partial MPI trace generated.

5. SCHEDULING WAITANY/TESTANY

ISP implements the POE algorithm that allows executing

```
1 : if (rank == 0)
2 : { MPI_Irecv(buffer, from 1, &req);
3 :   MPI_Barrier (MPI_COMM_WORLD);
4 :   MPI_Wait (&req, &status); }
5 : else if (rank == 1)
6 : { MPI_Isend(buffer1, to 0, &req);
7 :   MPI_Wait (&req, &status);
8 :   MPI_Barrier (MPI_COMM_WORLD); }
```

Figure 6: Sending large data with MPI_Isend

MPI operations out of the actual program order. Hence, when ISP traps an MPI request such as

```
MPI_Irecv (buffer, count, datatype, source, mpi_request).
```

ISP stores the arguments for later issue. Also, the argument `mpi_request` is stored in a hash map for a later retrieval. The key of the hash map is the request and the value is the corresponding `MPI_Irecv` or `MPI_Isend`.

Let *op* be an MPI operation. When *op* belongs to the set {`MPI_Wait`, `MPI_Waitall`, `MPI_Test`, `MPI_Testall`},

the out-of-order issue does not cause any problems since the POE algorithm's intra-completes before edges ensure that all ancestors *i.e*, the `MPI_Isend`s and `MPI_Irecv`s corresponding to the requests are issued before *op* itself is actually issued.

When *op* is in {`MPI_Testany`, `MPI_Waitany`}, all `MPI_Irecv` and `MPI_Isend` ancestors of *op* are *not necessarily* issued before *op* itself is issued. It is sufficient for just one of the `MPI_Isend` or `MPI_Irecv` operations corresponding to the requests to be completed in order for *op* to be issued. Let *op* have n requests and let $i \geq 1$ of them be completed (*i.e.*, i sends and/or receives have been issued to the MPI runtime). The MPI runtime is aware of only the i requests and has no knowledge of the rest of the $n - i$ requests as they were never issued to the MPI runtime. Hence, when ISP invokes *op* with all the n requests, an error is thrown by the MPI runtime that the request structure is invalid (which is true since MPI runtime is supposed to know n requests but it knows only i of them)!

In order to circumvent the above problem, ISP does the following:

1. For the requests that have been completed, remove the MPI requests from the hash map.
2. For each of the requests that is not completed, set the request to `MPI_REQUEST_NULL`. The requests with `MPI_REQUEST_NULL` are ignored by the MPI runtime.

This allows the `MPI_Testany` and `MPI_Waitany` to work with POE's out of order issue when the MPI runtime does not know ALL the requests it is supposed know during an in order execution.

6. SCHEDULING VS. MESSAGE SIZE

We first encountered this issue while running the Game of Life example mentioned earlier. As previously mentioned, the POE algorithm issues MPI operations to the MPI runtime only when a match-set is formed. Consider the example in Figure 6 where the MPI_Irecv in line 2 is matched with the MPI_Isend in line 6. ISP issues these two MPI functions to the MPI runtime. The MPI_Isend returns when the data in `buffer1` is copied into the MPI system buffer. However,

```
1: #include "mpi.h"
2: int MPIAPI MPI_Recv (void *buffer, int count,
3:     MPI_Datatype datatype, int source, int tag,
4:     MPI_Comm comm, MPI_Status *status) {
5:  static int num_calls = 0;
6:  int rank;
7:  num_calls++;
8:  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9:  if (rank == 0 && num_calls == 1)
10      return PMPI_Recv(buffer, count, datatype,
11                  2, tag, comm, status);
12  else
13      return PMPI_Recv(buffer, count, datatype,
14                  source, tag, comm, status);
15: }
```

Figure 7: Visual Studio Integration Method Illustrated

for large data it is possible that the MPI_Isend blocks until a matching receive starts receiving the message. Since POE issues both the MPI_Isend and MPI_Irecv to the MPI run time, the `MPI_Wait` in line 7 is expected to return immediately. However, when running the "Game of Life" MPI example, we encountered a similar situation as above (the above code is simplified and slightly modified to show the issue more clearly). In this case, the `MPI_Wait` in line 7 never returned!

This was because the MPICH implementation performs a "lazy receive" (still correct as per the MPI semantics) where it starts receiving *only when it encounters an* `MPI_Wait` or `MPI_Test` or when the control is given back to the MPI runtime that would force progress.

As a consequence, in the above example, process $P1$ would be stuck on the `MPI_Wait` of line 7. The POE algorithm does not issue `MPI_Barrier` since the match set is not formed yet. This means the for process 0, the control never gets back to MPI runtime. Hence, the MPI runtime does not make any attempt to progress process 0.

We overcome this problem by issuing an `PMPI_Wait` immediately after issuing `PMPI_Irecv`. When issuing the `PMPI_Wait`, we also store the `MPI_Status`. When the actual `MPI_Wait` of line 4 is issued, we just update the status field without issuing the `MPI_Wait` anymore.

7. USER INTERFACE DESIGN

We now discuss how the user interface of a Visual Studio integration of POE works. In general, we must strive to preserve the users' view of their MPI program, despite the fact that our POE algorithm changes the internal computation through dynamic rewriting of the wildcard receives.

The Verification Environment Visual Studio Add-in uses ISP by linking the MPI program with the ISP Profiler, and then runs the program through the command line Windows port of ISP. If a deadlock is found, the ISP output will include an error trace containing the MPI functions that were executed in order to reproduce the error. The Visual Studio Add-in is able to step Visual Studio's MPI Cluster Debugger to the error by analyzing the error trace and determining which process rank executes various MPI functions. The Add-in will step the debugger until the MPI function in ISP's error trace is executed, and it context switches between the processes as indicated by the error trace.

As described earlier, ISP's POE algorithm re-writes wild-

card receives, thus making a specific `MPI_Recv` match a specific `MPI_Send`. When debugging with the Visual Studio Add-in, it only has access to the ISP error trace, and it no longer is able to use the ISP Profiler which provides the wildcard re-writing. To solve this problem, the Add-in generates an additional file which is added to the Visual Studio project (shown in Figure 7). Analyzing the ISP error trace, the Add-in determines the process rank of the `MPI_Send` that matches the `MPI_Recv` and substitutes the correct rank instead of `MPI_ANY_SOURCE`. This additional file will force the Visual Studio debugger to follow the error trace correctly, instead of the possibility of matching any `MPI_Send` currently in the system. This technique is also used for `MPI_Irecv`, and a similar approach will need to be used with other rewritten functions (such as `MPI_Waitany` and `MPI_Waitsome`) in future versions of this Visual Studio interface.

8. CONCLUDING REMARKS

Dynamic verification is expected to be an essential part of future concurrent software debugging methods, mainly because there are too many APIs and libraries out there, and building models for them is not viable – at least in the short term. However, the price one pays for this luxury of using the "runtime as the transition system engine" is that occasionally the runtime ends up "having its own mind" when it comes to scheduling actions. The user will, in this case, have to make suitable modifications to the actual sequences of calls issued into the runtime. The overarching goal is often to avoid making intrusive modifications to APIs such as the MPI library.

We are in the process of designing a dynamic verification system for OpenMP. We are also in the process of handling MPI with multiple threads. Both these projects will require many more scheduling tricks to be invented. For example, with OpenMP, we plan to backtrack the entire code, however avoiding replaying interleavings within the OpenMP runtime. For MPI processes with multiple threads, the cleanest solution seems to be to rewrite those threads as extra MPI processes.

What if intrusive modifications to APIs are possible? With MPI, one can imagine a runtime to which we can issue a wildcard receive with an extra piece of information that says

> These sends have already been matched; please produce the remaining matches.

Such changes are not that difficult to make. In general the learning process appears to be the following progression of steps:

- Encounter the scheduling issues; solve them as we have been, through situation-specific solutions;
- Implement, evaluate, seek common mechanisms (or mechanisms that share common aspects);
- Gradually educate the API builders of future APIs as to the issues we are facing and the importance of avoiding some of the issues in the API design process. This may encourage future designers of APIs to incorporate better scheduling mechanisms into their designs to facilitate dynamic verification.

9. REFERENCES

[1] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[2] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000.

[3] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 110–121. ACM, 2005.

[4] Madan Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

[5] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Lecture Notes in Computer Science*, volume 4595, pages 58–75, 2007.

[6] http://research.microsoft.com/projects/CHESS/.

[7] Koushik Sen and Gul Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[8] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In *Computer Aided Verification*, 2008. Accepted.

[9] 2007. Personal conversation with Shaz Qadeer.

[10] http://www.cs.utah.edu/formal_verification/padtad08.

[11] Bettina Krammer and Michael M. Resch. Correctness checking of MPI one-sided communication using marmot. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI), LNCS 4192*, pages 105–114, 2006.

[12] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[13] Marc Snir and Steve Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[14] William D. Gropp and Ewing Lusk. Using mpi-2: A problem-based approach, 2007. Tutorial.

[15] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] Salman Pervez, Robert Palmer, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Practical model checking method for verifying correctness of mpi programs. In *EuroPVM/MPI*, pages 344–353, 2007.