

Build Dynamic Verifiers for Real Concurrency APIs, and Novel GUIs to Visualize Concurrency Nuances *

Ganesh Gopalakrishnan¹

School of Computing, Univ. of Utah, Salt Lake City, UT 84112, USA,
http://www.cs.utah.edu/formal_verification/

Acknowledgements: This position statement covers joint work with the students of the Gauss research group, principally Sarvani Vakkalanka, Anh Vo, Sriram Ananthakrishnan, and Michael DeLisi. Discussions with faculty CoPI Robert M. Kirby are gratefully acknowledged.

Summary: This paper is first a plea that formal methods researchers start studying and building support for the plethora of concurrency APIs being proposed. Formal methods researchers have shown heightened interest on the topic of transaction memories. However, a corresponding level of interest on other APIs (elaborated below) is lacking. Second, this paper explores ideas on how GUIs for formal methods tools applicable to real world concurrency APIs might differ from other GUIs (such as for traditional debuggers). While many ideas, such as message sequencing charts, have been pioneered in tools such as SPIN [1], each proposed concurrency API has enough nuances of concurrency, resource usage, communication, and synchronization that every attempt must be made to display them in an intuitively appealing manner.

This paper elaborates on these statements, and then walks the reader through these features of ISP [5, 6], our push-button dynamic formal verifier of Message Passing Interface [3] (MPI) programs. ISP has a graphical user interface (GUI) that is particularly tailored to give users insights into the message passing concurrency of MPI.

Details: A large number of APIs, domain specific languages, and annotation frameworks (generically called “APIs” in this paper) are under active consideration for the development of future concurrent systems. They include both well established APIs as well as some new ones; a partial list would be MPI, Intel Thread Building Blocks, various transaction memories (TM), OpenMP, CUDA, OpenCL, Parallel FX, Task Parallel Library, and MCAPI. One should begin educating students on the existence and state of maturity of these APIs. Current textbooks and descriptive papers do not emphasize the formal aspects of these APIs.

There must ideally be push-button dynamic formal verifiers for programs written against concurrency APIs. The importance of dynamic formal verification stems from the fact that they combine the best features of model checkers and traditional debuggers. In particular, they are easy to use, are based on concrete execution of actual codes (and not verification models), do not generate false alarms, and can generate counterexamples when assertions fail. As

* Supported in part by Microsoft, CNS-00509379, CCF-0811429, and SRC 1847.001

an example, ISP can be run like a debugger. It also provides superior coverage guarantees compared to traditional debuggers (see [2]).

There must be intuitive graphical user interfaces (GUIs) that can help visualize not only the ‘raw’ traces of execution, but also meaningful higher level aspects of execution including (for illustration) the Mazurkeiwicz traces, weak execution semantics, key aspects of non-determinism, other aspects of the communication/synchronization behavior that are likely to be misunderstood. We have built such a GUI for ISP which is detailed later in this paper.

Illustration: Our suggestions are illustrated on MPI, a true standardization success story. MPI runs on parallel machines of all sizes and scales. Virtually all the parallel simulation programs in science and engineering are written in MPI. According to Geist [4], the long term viability of MPI is ensured by the many mature MPI applications whose porting to other (more modern) notations is economically infeasible. Unfortunately, the MPI library is very large (over 300 functions), with at least 100 of these pertaining to concurrent communication and synchronization. Learning MPI through today’s available pedagogical material is highly unsatisfactory for anyone who cares about formal methods, as most textbooks and tutorials are written with an ad-hoc testing emphasis. Today’s available MPI debuggers that can directly run MPI sources do an excellent job of “heavy lifting” (handling large MPI programs, showing executions of tens of thousands of processes) but a poor job of covering the concurrency space, as well as highlighting aspects of concurrent executions for better debugging support. Our real proposal is how effectively to embed formal dynamic verification tools into existing verification frameworks, and help users to easily transition between high level and low level debugging.

ISP falls under the category of dynamic formal verification tools which were pioneered by Godefroid in the Verisoft [12] project. In MPI programs, most of the actions are dependent; however, when dependencies arise, they are a function of the execution state, and must be accurately handled to avoid interleaving explosion. For these reasons, dynamic partial order reduction (DPOR) [14] is a very natural fit for enhancing MPI dynamic verification. ISP employs an MPI-customized dynamic partial order reduction algorithm called Partial Order avoiding Elusive interleavings (POE) [5]; without POE, ISP would not be able to handle even the simplest of examples. Other tools (*e.g.*, CHESS [15]) employ other methods for interleaving reduction, such as preemption bounding. ISP can be downloaded [7] to run on Windows, MAC OS/X, and Linux, handles several MPI libraries, and has been very effective in finding bugs in large programs [6]. Here is a summary of features of ISP:

- ISP’s user interface generates an offline display based on a trace file. Unlike with conventional testing tools for MPI, the size of the trace file is potentially much smaller because POE generates, stores, and displays only the *relevant interleavings*. This helps discern bugs easily.
- For alternate wildcard matches, ISP uses intuitive graphics (dotted lines for alternative matches). This way, one can visualize non-deterministic choices without “display explosion.”

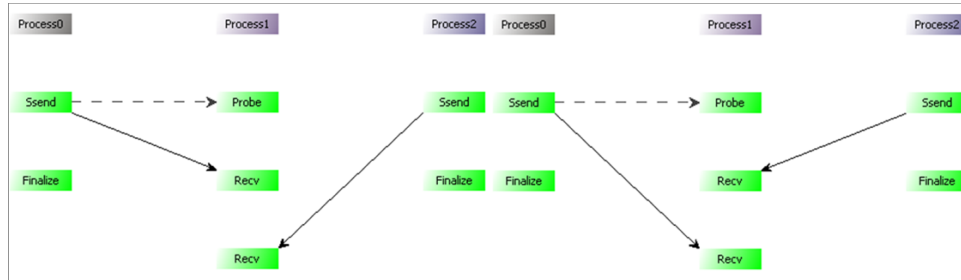


Fig. 1. Probe Non-determinism De-Mystified

- ISP has been integrated into the Microsoft Visual Studio framework. From there, a Java based GUI can also be launched. This GUI displays MPI function calls in iconic form, revealing the details of the calls through tool tips. All this functionality is being ported to become available within the Eclipse Parallel Tools Framework.
- The Visual Studio interface displays all communication matches, opening windows dynamically based on the number of matches to be shown. It displays deadlocks and other assertion violations graphically.
- The GUI allows the user to cut into the underlying Visual Studio debugger at any selected highlight point (e.g., communication match or deadlock). This allows smooth transition into low-level debugging when needed.
- The GUI includes a novel display of *completes-before* (CB) which helps designers understand how issued MPI commands may complete out of order – a *partial order that summarizes platform-specific scheduling variations*. The CB relation is fundamental to the POE algorithm [5].
- The CB relation also helps ISP determine whether certain MPI barrier statements are *functionally irrelevant* [17], that is, their removal does not alter the executions.¹
- For MPI Probes, ISP’s user interface displays which sends were probed and which were subsequently matched by MPI receives. It is well known that those sends that are probed need not be the ones from which subsequent receives perform receive operations. Figure 1 shows this occurring in an example (not shown) verified using ISP.
- Not all MPI collective operations *e.g.*, *broadcast* possess the barrier semantics. This fact is intuitively brought out by the CB graph.

1 Details

We now provide a walk-through of many simple examples that bring out these features of ISP. Consider the MPI example below where we highlight the MPI

¹ MPI barriers are often used for ‘good measure’ by programmers who cannot argue whether their removal is safe; in other cases, they are used to streamline input/output accesses.

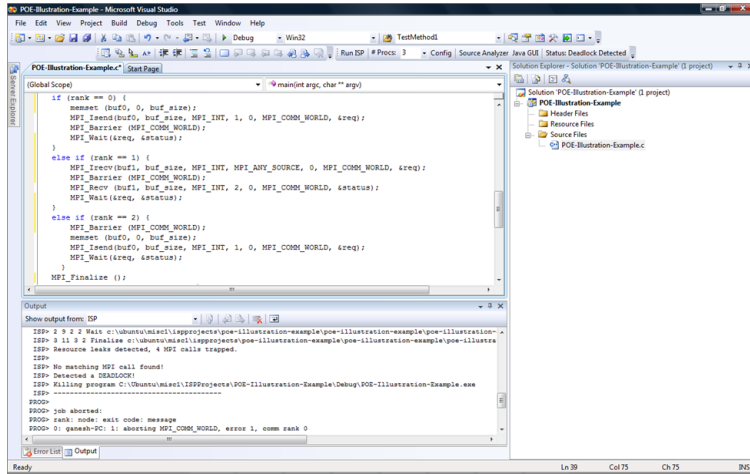


Fig. 2. Visual Studio Plug-in of ISP

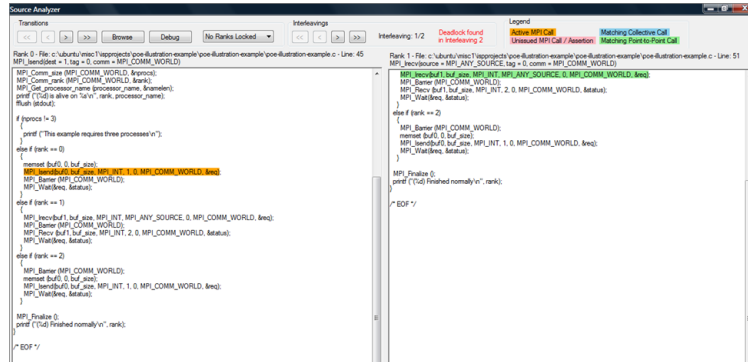
communication sources and targets and the communication handles. A user wanting to formally verify this program under ISP can launch the Visual Studio Plug-in of ISP, displaying the figure shown in Figure 2. The user can fire ISP from within this framework.

```
P0: Isend(to 1, &h) ; Barrier() ; Wait(&h) ; ...
P1: Irecv(from *, &h); Barrier() ; Recv(from 2); Wait(&h); ...
P2: Barrier() ; Isend(to 1, &h); Wait(&h) ; ...
```

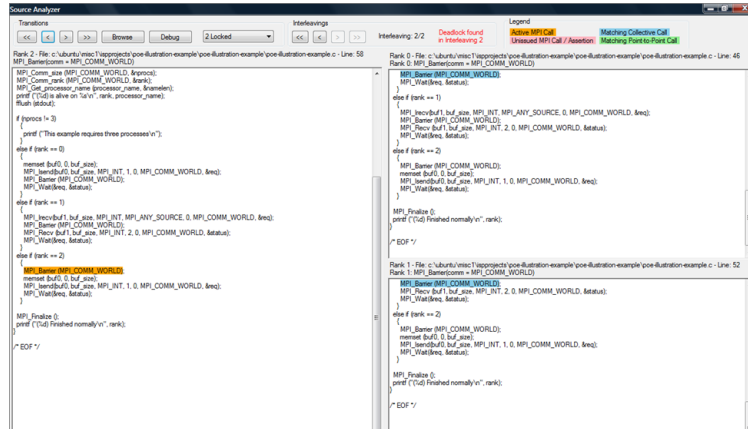
In this example, ISP will determine that the `Irecv` of P1 has two potential send matches. It then replays the program over two interleavings, forcing these send-recv matches. One of these interleavings results in a deadlock. The user can single-step the execution trace, obtaining the display shown in Figure 3(a). When MPI Barrier is encountered, the display shows all processes participating in the barrier (Figure 3(b)). Soon the user reaches the deadlock point, at which time the user invokes the transition navigator, obtaining a tree display of the transitions invoked (Figure 3(c)). Those transitions that are not matched yet are highlighted in red color in this display (and in pink on the left-hand side). The user may cut into Visual Studio for more incisive debugging (Figure 4(a)).

At this point, the user may want to root-cause the bug. They display the communication matches (Figure 4(b)) which shows the two interleavings that this example has. The wildcard receive – source of non-determinism – is shown in red. The interleaving in which P0’s `Isend` matches this receive (left) does not lead to a deadlock. The interleaving in which P2’s `Isend` matches this receive (right) does lead to a deadlock. The tool tips show the details of each icon (MPI command, file/line).

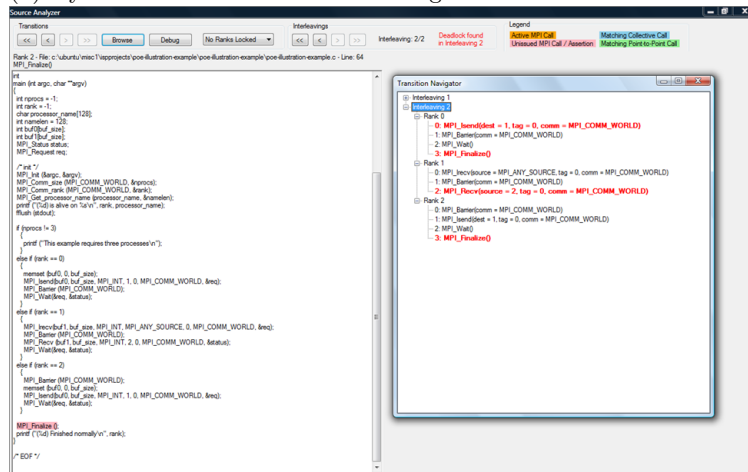
The user further wants to drill down and locate the bug. They display the full completes-before graph with respect to a chosen group of nodes (Figure 4(c)),



(a) Send/Receive Matches and Deadlock Display

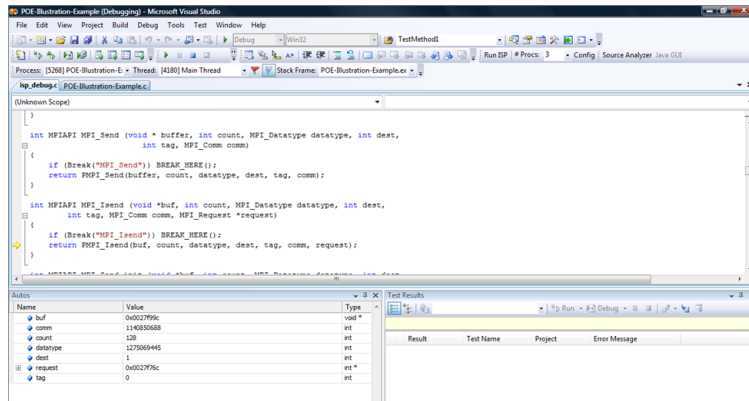


(b) Dynamic Window Creation Showing Collectives

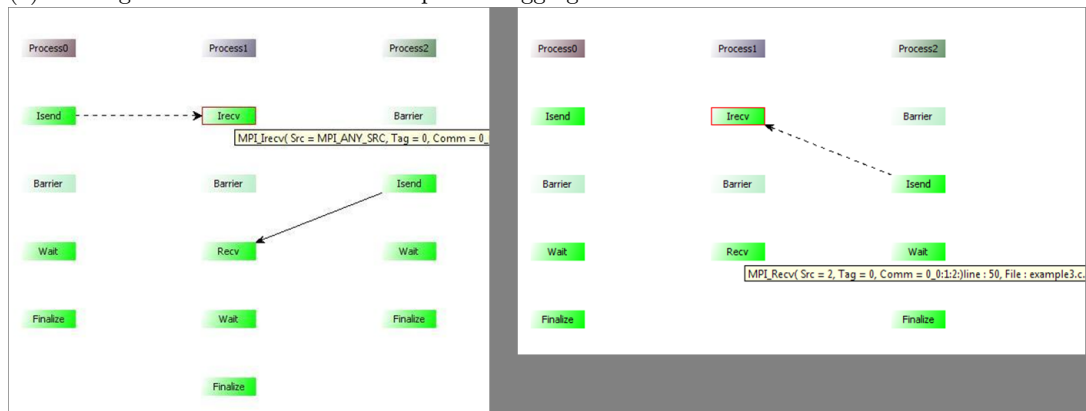


(c) Transition Navigator Display

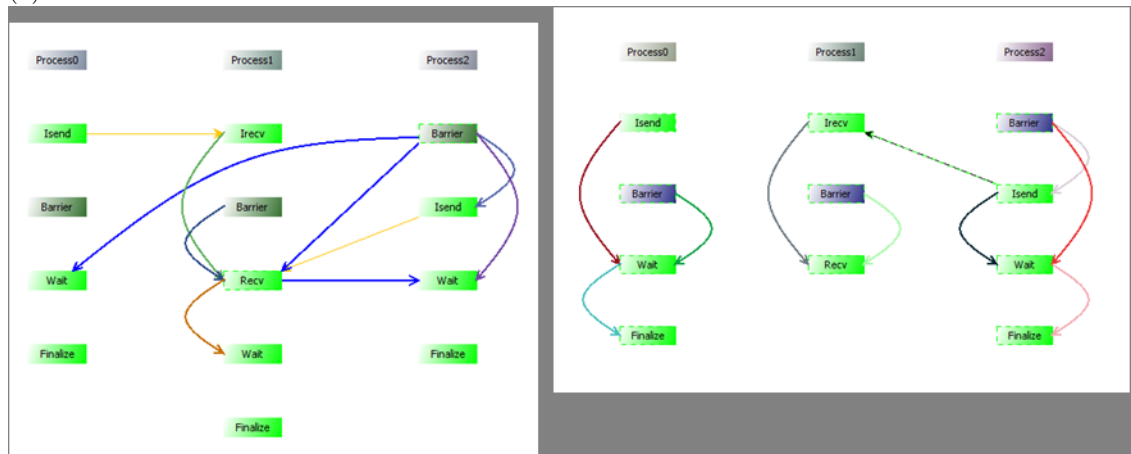
Fig. 3.



(a) Cutting into Visual Studio for Deeper Debugging



(b) Communication Matches



(c) Completes-before

Fig. 4.

left). They may merely display the *intra* completes-before relation (Figure 4(c), right) that shows the completes-before graph corresponding to the deadlocking interleaving. The completes-before graph shows that an `Isend` issued before an MPI barrier need not complete before the barrier; it can complete afterward! This tells the user that both wildcard matches are possible. The CB graph also tells the user how different platforms may process the MPI commands out of order. This tour should tell the reader that ISP’s MPI-specific GUIs are able to say a lot within a small amount of space. *In our experience, such GUIs can help newcomers understand the value of formal ideas.*

How CB is determined: The MPI standard requires non-overtaking in the sense that two MPI messages send from P_i and P_j arrive in that order (per tag and communicator). Thus, if there is a `Send(to P1)` followed by another `Send(to P1)` in an MPI program, these commands must *complete* in order (of course they are always issued in program order). On the other hand, if a `Send(to P1)` is followed by a `Send(to P2)` in an MPI process, these sends may finish out of order. Imagine the first one sending a gigabyte while the second one sends a byte; in this case, it would be inefficient to wait for the first send to finish. It also is no violation of non-overtaking to allow the second send to finish. The full definition of the CB relation appears in [5]. Our formalization of CB allows us to achieve two objectives at once: (i) schedule actions within POE so that the maximal extend of wildcard receive non-determinism is discovered (the same idea is also followed for wildcard probes); (ii) it also displays to the MPI user how each platform may reorder the commands issued from *the very same* MPI process. Of course, the display of relevant interleavings by ISP avoids exploding the view with a display of interleaving variations of commands issued from *different* processes.

Conclusions: We described our formal dynamic verification tool ISP. We believe that due to the wealth of examples and the intuitive GUI offered in ISP’s distribution, it is an ideal tool for teaching MPI as well as for serious development of large-scale MPI programs. To emphasize this point, we have recently ‘solved’ using ISP a majority of the exercises that appear in a widely used MPI textbook [18].

It would be good to see concurrency APIs being pursued actively by formal methods researchers. While there is certainly a lot of complexity in real APIs, and clean solutions (such as in futuristic proposals) may not be obtainable, this certainly gets practitioners involved in the area of Computer Aided Verification, as has been our experience. This, in a sense, is what EC2 is about.

References

1. Gerard J. Holzmann, “The SPIN MODEL CHECKER,” ADDISON-WESLEY, 2004.
2. ISP TESTS ON THE UMPIRE TEST SUITE. [HTTP://WWW.CS.UTAH.EDU/FORMAL_VERIFICATION/ISP_TESTS/](http://www.cs.utah.edu/formal_verification/isp_tests/).
3. MPI: A MESSAGE-PASSING INTERFACE STANDARD, [HTTP://WWW.MPI-FORUM.ORG/](http://www.mpi-forum.org/).
4. AL GEIST. MPI MUST EVOLVE OR DIE. KEYNOTE, EUROPVM/MPI 2008.

5. SARVANI VAKKALANKA, GANESH GOPALAKRISHNAN, AND ROBERT M. KIRBY. DYNAMIC VERIFICATION OF MPI PROGRAMS WITH REDUCTIONS IN PRESENCE OF SPLIT OPERATIONS AND RELAXED ORDERINGS. CAV 2008. PAGES 66-79. LNCS 5123.
6. ANH VO, SARVANI VAKKALANKA, MICHAEL DELISI, GANESH GOPALAKRISHNAN, ROBERT M. KIRBY, , AND RAJEEV THAKUR. FORMAL VERIFICATION OF PRACTICAL MPI PROGRAMS. *Principles and Practices of Parallel Programming (PPoPP)*, PAGES 261–269, 2009.
7. MICHAEL DELISI. [HTTP://WWW.CS.UTAH.EDU/FORMAL_VERIFICATION/ISP-RELEASE](http://www.cs.utah.edu/formal_verification/isp-release).
8. STEPHEN F. SIEGEL AND ANDREW R. SIEGEL. MADRE: THE MEMORY-AWARE DATA REDISTRIBUTION ENGINE. EURO PVM/MPI 2008. LNCS 5205. PAGES 218-226.
9. [HTTP://WWW.MPIBLAST.ORG](http://www.mpiblast.org).
10. ASYNCHRONOUS DIFFUSIVE LOAD BALANCING. [HTTP://WWW.ND.EDU/~CGAU/PARALLEL-PAPER/NODE14.HTML](http://www.nd.edu/~CGAU/parallel-paper/node14.html).
11. IRS: IMPLICIT RADIATION SOLVER 1.4 BUILD NOTES. LAWRENCE LIVERMORE NATIONAL LABS.
12. PATRICE GODEFROID. MODEL CHECKING FOR PROGRAMMING LANGUAGES USING VERISOFT. IN *POPL '97*. PAGES 174–186, 1997.
13. STEPHEN F. SIEGEL. THE MPI-SPIN WEB PAGE. [HTTP://VSL.CIS.UDEL.EDU/MPI-SPIN](http://vsl.cis.udel.edu/mpi-spin), 2008.
14. CORMAC FLANAGAN AND PATRICE GODEFROID. DYNAMIC PARTIAL-ORDER REDUCTION FOR MODEL CHECKING SOFTWARE. IN *POPL '05*, PAGES 110–121, 2005.
15. MADAN MUSUVATHI AND SHAZ QADEER. ITERATIVE CONTEXT BOUNDING FOR SYSTEMATIC TESTING OF MULTITHREADED PROGRAMS. PLDI, PAGES 446–455, 2007. ALSO SEE [HTTP://RESEARCH.MICROSOFT.COM/CHES](http://research.microsoft.com/chess)S.
16. SARVANI VAKKALANKA, MICHAEL DELISI, GANESH GOPALAKRISHNAN, AND ROBERT M. KIRBY. SCHEDULING CONSIDERATIONS FOR BUILDING DYNAMIC VERIFICATION TOOLS FOR MPI. PADTAD 2008.
17. SUBODH SHARMA, SARVANI VAKKALANKA, GANESH GOPALAKRISHNAN, ROBERT M. KIRBY, RAJEEV THAKUR, AND WILLIAM GROPP. A FORMAL APPROACH TO DETECT FUNCTIONALLY IRRELEVANT BARRIERS IN MPI PROGRAMS. EURO PVM/MPI, PAGES 265–273, 2008. LNCS 5205.
18. PETER PACHECO. *Parallel Programming with MPI*. MORGAN KAUFMANN, 1997.