

# SPINning Parallel Systems Software<sup>\*</sup>

Olga Shumsky Matlin, Ewing Lusk, and William McCune

Mathematics and Computer Science Division  
Argonne National Laboratory  
Argonne, IL 60439, USA  
`{matlin,lusk,mccune}@mcs.anl.gov`

**Abstract.** We describe our experiences in using SPIN to verify parts of the Multi-Purpose Daemon (MPD) parallel process management system. MPD is a distributed collection of processes connected by Unix network sockets. Its dynamic nature is easily expressible in the SPIN/PROMELA framework but poses performance and scalability challenges. We present here the results of expressing some of the parallel algorithms of MPD and executing verification runs with SPIN.

## 1 Introduction

Reasoning about parallel programs is surprisingly difficult. Even small parallel programs are difficult to write correctly, and an incorrect parallel program is equally difficult to debug, as we experienced while writing the Multi-Purpose Daemon (MPD), a process manager for parallel programs [1,2]. Despite MPD's small size and apparent simplicity, errors have impeded progress toward code in which we have complete confidence. Such a situation motivates us to explore program verification techniques. In our first attempt [9], based on the ACL2 [7] theorem prover, formulating desired properties of and reasoning about models of MPD algorithms proved difficult. Our second approach employs the model checker SPIN [6].

MPD is itself a parallel program. Its function is to start the processes of a parallel job in a scalable way, manage input and output, handle faults, provide services to the application, and terminate jobs cleanly. MPD is the sort of process manager needed to run applications that use the standard MPI [10,11] library for parallelism, although it is not MPI specific. MPD is distributed as part of the portable and publicly available MPICH [4,5] implementation of MPI.

The remainder of the paper is structured as follows. In Section 2 we describe MPD in more detail and outline our method for modeling a distributed, dynamic set of Unix processes in PROMELA. In Section 3 we present the concrete results of specific verification experiments. We conclude in Section 4 with a summary of the current project status and our future plans.

---

<sup>\*</sup> This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

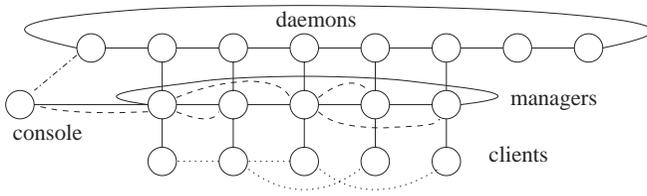


Fig. 1. Daemons with console process, managers, and clients

## 2 Approach

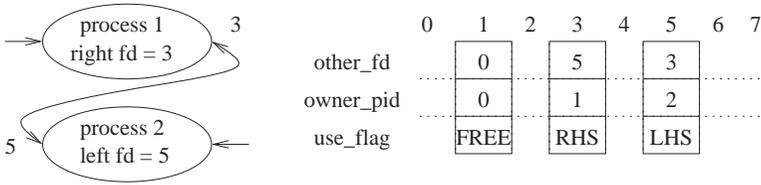
The MPD system comprises several types of processes. The *daemons* are persistent (may run for weeks or months at a time, starting many jobs) and are connected in a ring. *Manager* processes, started by the daemons to control the application processes (*clients*) of a single parallel job, provide most of the MPD features and are also connected in a ring. A separate set of managers supports an individual process environment for each user process. A *console* process is an interface between a user and the daemon ring. A representative topology of the MPD system is shown in Figure 1. The vertical solid lines represent connections based on pipes; the remaining solid lines all represent connections based on Unix sockets. The remainder are potential or special-purpose connections.

Each of the daemon, manager, and console process types has essentially the same pattern of behavior, which is important for our purpose. After initialization, the process enters an infinite, mostly idle, loop, implemented by a Unix socket function `select`. When a message arrives on one of its sockets, the process calls the appropriate message handler routine and reenters the idle `select` state. The handler does a small amount of processing, creating new sockets or sending messages on existing ones. The logic of the distributed algorithms executed by the system as a whole is contained primarily in the handlers, and this is where the difficult bugs appear.

### 2.1 Modeling Components of the Multi-purpose Daemon

Components of the MPD system map naturally to PROMELA entities: a `proctype` is defined for each MPD process type, and sockets are represented by channels. The structure of the MPD process types allows us to treat them as comprising three tiers. The top tier corresponds to the upper-level sequential logic of the process (initialization, `select` loop, calling the handlers). Handlers form the second tier. The bottom tier corresponds to well-understood Unix socket operations.

An MPD programmer uses, but does not implement, predefined libraries of the socket functions. However, having to model the socket operations explicitly, we created a PROMELA library of the primitives, which allows us to (1) hide the details of the socket model from both the verification and the future mapping of the model to the executable code, (2) interchange, if need be, different models of sockets without changing the remainder of the model, and (3) reuse the socket model in verifying independent MPD algorithms.



**Fig. 2.** Connected MPD processes and corresponding socket records array

## 2.2 Modeling Unix Sockets

A Unix socket, an endpoint of a bidirectional communication path, is manipulated when a connection between two processes is established or destroyed. A socket is referenced by a file descriptor (fd) and represents a buffer for reading and writing messages. Our PROMELA model of a socket consists of a channel and a three-part record. The first record field references the fd at the other endpoint of the connection. The second field identifies a process that has an exclusive control over the socket. The third field indicates whether the socket is free and, if not, how it can be used by the owner process. Figure 2 shows two connected processes and the corresponding state of an array of the socket descriptors.

Five Unix socket primitives have been modeled according to [13]: `connect`, `accept`, `close`, `read`, and `write`. As `select` appears only in the top tier of the MPD process, its implementation is specific to an MPD algorithm. Below is an excerpt of the socket library.

```

typedef conn_info_type {                               /* socket descriptor */
    unsigned other_fd : FD_BITS;
    unsigned owner_pid : PROC_BITS;
    unsigned use_flag : FLAG_BITS;
};
conn_info_type conn_info[CONN_MAX];
chan connection[CONN_MAX] = [QSZ] of {msg_type};

inline read(file_desc, message) {
    connection[file_desc]?message; }

inline write(file_desc, message) {
    connection[conn_info[file_desc].other_fd]!message;
    fd_select_check(conn_info[file_desc].other_fd) }

inline close(file_desc) {
    IF /* other side has not been closed yet */
    :: (conn_info[file_desc].other_fd != INVALID_FD) ->
        set_other_side(conn_info[file_desc].other_fd, INVALID_FD);
        fd_select_check(conn_info[file_desc].other_fd)
    FI;
    deallocate_connection(file_desc) }

```

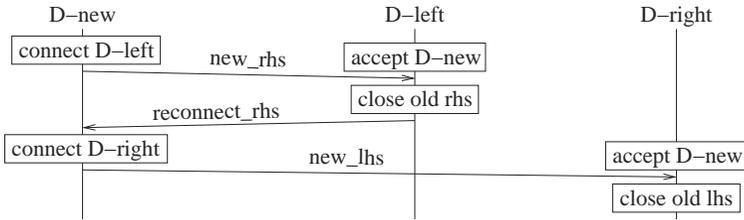


Fig. 3. Parallel ring insertion algorithm

```

inline connect(file_desc, lp) {
    allocate_connection(j);          /* server's connection */
    set_owner(j, lp);                /* finalized by accept */
    set_handler(j, AWAIT_ACCEPT);
    allocate_connection(file_desc); /* client's connection */
    set_owner(file_desc, _pid);
    set_other_side(j, file_desc);    /* relate connections */
    set_other_side(file_desc, j);    /* to each other */
    lp_select_check(lp)              }

inline accept(file_desc) {
    file_desc = 0;
    do /* next line is a simplification of real accept */
    :: (file_desc >= CONN_MAX) -> assert(0) /* error, no connect */
    :: (readable_lp(file_desc, _pid)) ->
        set_handler(file_desc, NEW); break
    :: else -> file_desc = file_desc + 1
    od }
  
```

### 3 Verification of MPD Algorithms

Most interesting MPD algorithms reside in the daemons and the managers. We modeled and verified algorithms for daemon ring creation and recovery and a manager-level barrier algorithm. The models conform to the three-tiered view of the algorithms and rely on the socket library. We used bit-arrays and followed recommendations from [12] and employed ideas from [3].

#### 3.1 Ring Establishment Algorithm

To create a daemon ring, the initial daemon establishes a listening port to which subsequent connections are made. A ring of one daemon is created. The other daemons enter the ring by connecting to the first daemon. Figure 3 shows an MSC representation of an algorithm that allows several daemons to enter the ring simultaneously. To establish the ring and to recover from a failure, each daemon

maintains identities of the two right-hand side neighbors (`rsh` and `rsh2`). Upon receipt of `new_rhs`, `D-left` sends a message of type `rsh2info` (not shown in the figure), counterclockwise along the ring, to notify its left-hand side neighbor that `D-new` is its new `rsh2`. Upon receipt of `new_lhs`, `D-right` sends `rsh2info`, also counterclockwise, to notify `D-new` that its `rsh2` is the `rsh` of `D-right`.

We verify that, upon algorithm completion, the resulting system topology implicit in the socket descriptor structures array is a ring of the correct size. We also check that in each daemon the identities of `rsh` and `rsh2` agree with the information in the array. These two conditions are collectively referred to as the *state* property. MPD designers and users test the ring by invoking `mpitrace`, which reports the identities and positions of all daemons. To convince our “customers” (i.e., MPD designers) of the correctness of our model, we model the trace algorithm and prove its termination. Because the algorithm involves sending additional messages, verification of the trace completion is more expensive, in time and memory, than verification of the state property.

### 3.2 Recovery from a Single Nondeterministic Failure

Upon daemon crash, the operating system will close all associated sockets, which will be visible to its neighbors. When the right-hand side socket of the daemon to the left of the crash is closed unexpectedly, the daemon reinstates the ring by connecting to its `rsh2` neighbor. In our model a random daemon in the initial hard-coded ring is directed to fail, and the recovery procedure is initiated. The model was verified against the state and trace termination correctness properties.

### 3.3 Manager-Level Barrier Algorithm

Parallel jobs (programs running on the clients) rely on the managers to implement a synchronization mechanism, called *barrier*. In our model of the algorithm some messages are abstracted, and the clients are not modeled explicitly. When a leader manager receives a barrier request from its client (represented by setting the `client_barrier_in` bit), it sends the message `barrier_in` along the ring. A non-leader manager holds the message (setting the bit `holding_barrier_in`) until its client requests the service. Once the `barrier_in` is received by the leader, the message is converted to `barrier_out` and sent along the ring. Upon receipt of `barrier_out`, managers notify the clients, setting the `client_barrier_in` bit, that the synchronization has occurred. A special constant `ALL_BITS` corresponds to a bit-vector with all bits set.

We verified two correctness conditions. First, all clients pass the barrier.

```
timeout -> assert(client_barrier_out==ALL_BITS)
```

Second, an invariant holds: a client is allowed to proceed only when all clients have reached the barrier and all managers have released the `barrier_in` message.

```
assert((client_barrier_out==0) ||
        ((client_barrier_in==ALL_BITS) && (holding_barrier_in==0)))
```

**Table 1.** Verification statistics summary

Algorithm	Property	Model Size	Time (s)	Memory (MB)	Vector Size (byte)	States Stored/Matched	Search Depth
Insert	State	4	105.35	768	224	3.83e+06/7.97e+06	115
Insert	Trace	3	0.80	3.3	136	5743/6718	58
		4*	159.33	173	224	4.57e+06/9.28e+06	115
Recover	State	9	69.70	376.0	520	734040/3.26e+06	158
		12*	5340.19	772.6	876	2.75e+07/1.76e+08	209
Recover	Trace	8	163.39	814.1	464	1.93e+06/7.83e+06	199
		9*	1919.94	502.5	520	1.62e+07/7.85e+07	232
Barrier		12	127.97	549.2	288	1.95e+06/1.18e+07	101
		14*	3050.96	571.3	332	1.75e+07/1.25e+08	117

### 3.4 Verification Statistics

Table 1 summarizes verification statistics for the three algorithms. All verification runs were conducted on a 933 MHz Pentium III processor with 970 MB of usable RAM, with default XSPIN settings, except the memory limit. Compression (`-DCOLLAPSE` compile-time directive) was used in cases identified by an asterisk. We show statistics for the largest models on which verification succeeded with and without compression.

We were unable to exhaustively verify the ring insertion algorithm on models with five or more daemons, even with compression. Applying predicate abstraction techniques enabled verification of models of eight daemons, but a desirable correlation of the model to the C code was lost, as was the ability to perform meaningful simulations. For other algorithms, verification succeeded for larger models, although, admittedly, the algorithms are rather simple.

Let us put in perspective the size of models on which verification succeeds. While a running MPD may consist of tens or hundreds of processes, prior experience with debugging MPD code suggests that even the most difficult errors manifest themselves in systems of four to ten processes. Therefore, the models of some MPD algorithms at the current level of abstraction allow us to verify some algorithms on models of satisfactory size. For other algorithms, such as the ring establishment algorithm, a slightly more abstract model or a more efficient socket library will enable meaningful verification.

## 4 Summary and Future Plans

We described here our initial experiences in applying the SPIN-based approach to verifying a parallel process management system called MPD. Our models relied on a reusable model of Unix socket operations. For the ring establishment algorithm, we were able to complete exhaustive verification only on models with up to four daemons. We were, however, able to exhaustively verify larger models of other algorithms.

```

p.1  :: (msg.cmd == barrier_in) ->
p.2    if
p.3    :: (IS_1(client_barrier_in,_pid)) ->
p.4      if
p.5      :: (_pid == 0) ->
p.6        make_barrier_out_msg;
p.7        find_right(fd,_pid);
p.8        write(fd,msg)
p.9      :: else ->
p.10       make_barrier_in_msg;
p.11       find_right(fd,_pid);
p.12       write(fd,msg)
p.13     fi
p.14   :: else ->
p.15     SET_1(holding_barrier_in,_pid)
p.16   fi

c.1  if ( strcmp( cmdval, "barrier_in" ) == 0 ) {
c.2    if ( client_barrier_in ) {
c.3      if ( rank == 0 ) {
c.4        sprintf( buf, "cmd=barrier_out dest=anyone src=%s\n", myid );
c.5        write_line( buf, rhs_idx );
c.6      }
c.7      else {
c.8        sprintf( buf, "cmd=barrier_in dest=anyone src=%s\n", origin );
c.9        write_line( buf, rhs_idx );
c.10     }
c.11   }
c.12   else {
c.13     holding_barrier_in = 1;
c.14   }
c.15 }

```

**Fig. 4.** Portion of the PROMELA model and C implementation of the barrier algorithm

Based on our experiences, we believe that design and development of algorithms for MPD and similar systems can benefit greatly from application of the SPIN-based software verification methods. SPIN's simulation capability enables rapid prototyping of new algorithms. Since even the most difficult errors can be discovered on models comprising only a few processes, the verification engine of SPIN enables us to verify the algorithms on models that are sufficiently large for our purposes.

A long-term goal of this project is to model and verify MPD algorithms and then translate them into C or another programming language, while preserving the verified properties. Ideally, translation should be automated. To allow this to happen, the PROMELA model must not be overly abstract. Figure 4 shows a PROMELA model and a C implementation of a portion of the barrier algorithm. Given the one-to-one correspondence between the control structures, and

a mapping between the rest of the code, automated translation certainly appears feasible for this level of abstraction. In general, verifiable models of the MPD algorithms should fall into just a few different classes with respect to the level of abstraction, and a separate mapping can be defined for each class to enable the PROMELA-to-C translation. Only the handlers in the middle tier of the process model need to be translated.

Many algorithms execute in parallel in a running MPD system, and their interaction is important. We hope to be able eventually to reason formally about MPD models that consist of several related and interdependent algorithms.

More information on the project, including the discussed PROMELA models, can be found in [8] and at <http://www.mcs.anl.gov/~matlin/spin-mpd>.

## References

1. R. Butler, W. Gropp, and E. Lusk. A scalable process-management environment for parallel programs. In J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS 1908, pages 168–175. Springer Verlag, September 2000.
2. R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27:1417–1429, 2001.
3. E. Fersman and B. Jonsson. Abstraction of communication channels in Promela: A case study. In K. Havelund, J. Penix, and W. Visser, editors, *Proceedings of the 7th International SPIN Workshop*, LNCS 1885, pages 187–204. Springer Verlag, 2000.
4. W. Gropp and E. Lusk. MPICH. <ftp://info.mcs.anl.gov/pub/mpi>.
5. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
6. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 22(5):279–295, May 1997.
7. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
8. O. S. Matlin, E. Lusk, and W. McCune. SPINning parallel systems software. Preprint ANL/MCS-P921-1201, Argonne National Laboratory, 2001.
9. W. McCune and E. Lusk. ACL2 for parallel systems software. In M. Kaufmann and J. S. Moore, editors, *Proceedings of the 2nd ACL2 Workshop*. University of Texas, 2000. <http://www.cs.utexas.edu/users/moore/acl2/workshop-2000>.
10. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
11. Message Passing Interface Forum. MPI2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.
12. T. C. Ruys. Low-fat recipes for SPIN. In K. Havelund, J. Penix, and W. Visser, editors, *Proceedings of the 7th International SPIN Workshop*, LNCS 1885, pages 287–321. Springer Verlag, 2000.
13. W. R. Stevens. *Unix Network Programming*, volume 1. Prentice Hall PTR, second edition, 1998.