MPI-Spin Version 1.1: A User's Manual

Stephen F. Siegel
The Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware

Contents

1. Introduction

1.1. **Background.** Spin [1, 2] is a popular open-source model checking tool originally developed at Bell Labs. This document assumes you are already familiar with Spin. To learn more about Spin, visit `http://spinroot.com`.

This version of the manual prepared on: June 7, 2008.

The *Message Passing Interface* (MPI) [3, 4] is a specification for a library of message-passing functions, with bindings in C, C++, and Fortran. There are numerous open-source and proprietary implementations of MPI. MPI is widely used to construct high-performance parallel programs, especially for scientific computation. This document assumes you are already familiar with MPI. For more information on MPI, see `http://www-unix.mcs.anl.gov/mpi` or `http://www.mpi-forum.org`.

MPI-Spin [5] is an extension to Spin that can be used to verify models of MPI programs. It adds to Spin's input language a number of functions, types, and constants that correspond closely to many of the primitives described in the MPI Standard.

1.2. **License.** MPI-Spin works with a (very slightly) modified version of Spin called Spin-M, which comes with its own copyright and license and must be downloaded separately. The instructions for dowloading and installing MPI-Spin can be found in §2.

MPI-Spin (but not Spin or Spin-M) is licensed under the GNU General Public License. In particular, MPI-Spin is distributed *without any warranty*; without even the implied warranty of *merchantability or fitness for a particular purpose.* See the file `LICENSE` for details.

2. Download and Installation

To both install and use MPI-Spin, the following must all be in your path: `cc`, `perl`, `which`, `make`, and `sh`. Assuming that is the case, the following process will install MPI-Spin in the directory `foo`:

1. Download, gunzip, and untar `mpi-spin`. Move `mpi-spin` into `foo`, if it isn't already there.
2. Download, gunzip, and untar `Spin-M`. (This is a (very slightly) modified version of Spin.) Move `Spin-M` into `foo/mpi-spin`.
3. Change into `foo/mpi-spin` (if you aren't already there) and type `./setup`. The setup script will try to determine your type of architecture and create an appropriate `local.mk` file. It does this by copying one of the files from the directory `arch`. If it can't figure out your architecture, you can specify the choice as a command line argument (e.g., `./setup linux` will copy `arch/linux.mk` to `local.mk`). These files are very simple and contain only a few paths and flags that should be used with your C compiler and pre-processor; you can also create one of your own, say `mysystem.mk`, place it in `arch`, and type `./setup mysystem`.
4. Type `make`. Everything should compile and build without warnings. If anything goes wrong, you might want to try some different paths/flags in the previous step.
5. There should now be a directory `foo/mpi-spin/bin` containing some executable files, including `ms` and `mscc`. Either add this directory to your path or copy or move the executable files to another directory in your path.

If you can't get something to work, email `siegel@cis.udel.edu` with as complete a description of the problem as possible.

3. The Structure of the Input File

An MPI-Spin input file is very much like an ordinary Spin input file, with a few slight differences, and a number of additional primitives corresponding to MPI primitives.

3.1. **Process type definitions.** A *process type definition* describes a process type, but does not instantiate any processes of that type. It has the form

```
BEGIN_MPI_PROC(name)
  .
  .
  .
END_MPI_PROC(name)
```

To instantiate and run a process of this type, one invokes `run name()`, just as one does for ordinary Spin processes. One may instantiate multiple processes of a single type through multiple calls to `run`.

An *active* process type definition has the form

```
BEGIN_ACTIVE_MPI_PROC(name, number)
  .
  .
  .
END_ACTIVE_MPI_PROC(name)
```

This is just like an ordinary process type definition, but it automatically instantiates `number` processes of this type in the initial state. There is no need to invoke `run`.

It is also necessary to instantiate and run an *MPI dæmon process*. This process is responsible for carrying out all the actions of the "MPI infrastructure", such as matching send and receive requests. This can be done in the active or inactive manner. In the active manner one simply inserts

```
ACTIVE_MPI_DAEMON
```

at some point in the global scope of the model. In the inactive manner one instead inserts

```
MPI_DAEMON
```

and then to run the process one invokes `RUN_MPI_DAEMON`. The inactive approach allows finer control over which PIDs Spin will assign to the processes, including the MPI dæmon process, as the PIDs are assigned in the order in which processes are instantiated.

3.2. **Example: Multiple-producer single consumer.** An example of the use of nonblocking communication is given in the following MPI/C code, which is extracted from [7, Ex. 2.18]:

```
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
if (rank != size-1) { /* producer code */
  while (1) {
    /* produce data */
    MPI_Send(buffer->data, buffer->datasize, MPI_CHAR, size-1 tag, comm);
  }
} else { /* consumer code */
  for (i=0; i < size-1; i++)
    MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
      &(buffer[i].req));
  for (i=0; ; i=(i+1)%(size-1)) {
    MPI_Wait(&(buffer[i].req), &status);
    /* consume data */
    MPI_Irecv(buffer[i].data, MAXSIZE, MPI_CHAR, i, tag, comm,
      &(buffer[i].req));
  }
}
```

In this program, multiple producers repeatedly send messages to a single consumer. The consumer posts receive requests for each producer in order of increasing rank, and then waits on each request in a cyclic order. After a receive request completes, the message is consumed and another receive request is posted for that producer. Note that overlap between computation and communication is achieved because the consumer may consume a message from a producer while the MPI infrastructure carries out the requests to receive data from other producers.

3.3. **A "communication skeleton" model.** An MPI-Spin model of this code, in which the data has been abstracted away entirely, is given as follows:

```
#include "mpi-spin.prom"

#if (NPRODUCERS != NPROCS - 1)
#error "Wrong NPRODUCERS"
#endif
#define TAG 0
#define SEND_BUFF NULL
#define RECV_BUFF NULL
#define COUNT 0

BEGIN_ACTIVE_MPI_PROC(producer, NPRODUCERS)
  MPI_Init(Pproducer, Pproducer->_pid);
  do
  :: MPI_Send(Pproducer, SEND_BUFF, COUNT, MPI_POINT, NPROCS - 1, TAG)
  od;
  MPI_Finalize(Pproducer)
END_MPI_PROC(producer)

BEGIN_ACTIVE_MPI_PROC(consumer, 1)
  MPI_Request req[NPRODUCERS];
  byte i = 0;

  MPI_Init(Pconsumer, Pconsumer->_pid);
  do
  :: i < NPRODUCERS ->
     MPI_Irecv(Pconsumer, RECV_BUFF, COUNT, MPI_POINT,
               Pconsumer->i, TAG, &Pconsumer->req[Pconsumer->i]);
     i++
  :: else -> i = 0; break
  od;
  do
  :: MPI_Wait(Pconsumer, &Pconsumer->req[Pconsumer->i],
              MPI_STATUS_IGNORE);
     MPI_Irecv(Pconsumer, RECV_BUFF, COUNT, MPI_POINT, Pconsumer->i,
               TAG, &Pconsumer->req[Pconsumer->i]);
     i = (i + 1)%NPRODUCERS
  od;
  MPI_Finalize(Pconsumer)
END_MPI_PROC(consumer)

ACTIVE_MPI_DAEMON
```

Some points to notice about the model are as follows.

First, it begins with a preprocessor directive to include the file `mpi-spin.prom`. This is the case for all MPI-Spin models.

An *active* MPI process type definition is used to define a collection of processes that are to start running automatically in the initial state (as opposed to having to be started by an explicit call to `run`). Such a definition is bracketed by `BEGIN_ACTIVE_MPI_PROC` and `END_ACTIVE_MPI_PROC` statements. The name of the process type and the number of processes of that type to instantiate are specified as arguments.

Also, notice that two different process types are defined, one for all of the producers and one for the sole consumer. One could put all the code into one process type definition, which would correspond more closely to the typical MPI style, but the way we have constructed the model is usually more efficient and more readable. The model also ends with the line `ACTIVE_MPI_DAEMON`. This is to declare and start the MPI dæmon process, which models all components of the MPI infrastructure. Something like this must be included in any model that will have MPI communication.

There are some differences between the syntax of the MPI functions of MPI-Spin and those for the C bindings of the MPI functions. First, for almost all of the MPI-Spin functions, the first parameter is the string consisting of the letter `P` followed by the proctype name. We will refer to this string as the *proc string*. Another difference arises from the way that C code is incorporated into Promela models. This leads to two different *contexts* in a model: the C context and the Promela context. Most of the parameters for the MPI-Spin versions of the MPI functions are in the C context. In order to refer to process-local Promela variables (e.g., `req`) in a C context, one must pre-append the proc string followed by `->` to the variable name. To refer to global Promela variables in the C context, pre-append `now.` to the variable name.

Notice that the MPI-Spin version of `MPI_Init` takes an integer argument. This is used to tell the MPI infrastructure what rank to use for this process. This differs from the usual MPI approach in which, conceptually, the MPI infrastructure assigns the rank and the user can only query it. In this example, we are just using the PID assigned to the process by Spin for the rank, which Spin stores in the process-local Promela variable `_pid`. That approach may in fact suffice for most examples, but there may be cases where one wishes to assign ranks in a way that differs from the way Spin assigns PIDs, and one should be free to do so. Basically, Spin assigns PIDs, starting at 0, to processes in the order in which they are started. In our model, since all processes are active, this is the same as the order in which they are declared. Hence the `NPRODCUERS` producer processes are assigned PIDs 0 through `NPRODUCERS` $-$ 1 and the consumer process is assigned PID `NPRODUCERS`.

Notice that the communication operations in the model use an MPI datatype `MPI_POINT`, which does not actually exist in MPI. This datatype is used to represent data that has been abstracted away completely ("to a point"). Conceptually, the number of bytes required to represent a "point" is 0. Nevertheless, the `count` argument used in sends and receives of points is relevant. So, for example, if one attempts to receive a message consisting of 10 points with a receive that specifies a maximum of 5 points, a buffer overflow error will be reported by MPI-Spin. The buffer argument is ignored for points, so we are free to use `NULL` for the send and receive buffers. This sort of abstraction is convenient when one wishes to capture only the MPI "communication skeleton" of the program and is not interested in the data itself.

At present, there is only one communicator in MPI-Spin, namely `MPI_COMM_WORLD`. This is why the `comm` argument has been left out of all the MPI-Spin versions of the MPI functions.

The syntax and semantics of the MPI-Spin functions, as well as a description of all the constants and types, is given in §5.

4. Executing MPI-Spin

One uses MPI-Spin to verify that one or more properties hold on all executions of an MPI-Spin model. The process for carrying this out is quite similar to the usual process for verifying properties of models with Spin. This typically consists of the following steps: (1) generating an analyzer from the model, (2) compiling the analyzer, and (3) executing the analyzer. If the result of execution of the analyzer reveals that the property can be violated, one might also want to (4) examine the trace.

4.1. **Generating the analyzer.** To generate the analyzer, one executes the script `ms`. In doing so, one must specify the number of MPI processes as well as other bounds.

In the *nonblocking* mode (which is the default mode, and can be used for any model), the user must specify upper bounds on (1) the total number of outstanding requests, and (2) the total number of buffered messages. If the request bound is exceeded during the search, an error is reported. The buffer bound is treated differently: buffering of messages is simply blocked whenever the total number of buffered messages equals the buffer bound.

An alternative *blocking* mode can be used with models that use only blocking MPI communication constructs. It is usually more efficient for such models. This mode is invoked with the command-line option `-block`. In this mode, the user must also specify an upper bound on the number of buffered messages that can exist for any ordered pair of processes. That bound is specified using the command-line option `-chansize=M`, where `M` is the integer literal upper bound.

The full set of options for `ms` can be obtained by simply executing `ms` with no arguments:

```
ms: generate mpi-spin verifier source code from Promela file
Usage: ms [options] <filename>
Options:
-np=<NPROCS>
   number of MPI processes (required)
-block
   use channel-based model; for models with only blocking communication
-chansize=<INT>
   channel size; required in blocking mode (-block)
-buf=<BUFFER>
   bound on number of buffered message (required in nonblocking mode)
-req=<REQUESTS>
   bound on number of outstanding requests (required in nonblocking mode)
-sym=[0|1|2]
   symbolic arithmetic: 0 (Herbrand, default), 1 (IEEE), or 2 (Real)
-symhash=<SYMBOLIC_HASHTABLE_LENGTH>
   hashtable length for symbolic values
-symmax=<SYMBOLIC_MAX_VALUES>
   bound on number of symbolic values
-crmax=<CR_MAX_VAL>
   bound on number of communication record values
-crhash=<CR_HASHTABLE_LENGTH>
   hashtable length for communication record values
-notest
   the model contains no MPI_Test*, MPI_*any*, nor MPI_*some*
-nocancel
   the model contains no MPI_Cancel
-noprobe
   the model contains no MPI_Iprobe nor MPI_Probe
-noanysource
   the model contains no MPI_ANY_SOURCE
-dl
   nondeterministically choose to synch/buffer sends for full
   deadlock detection
-v
   verbose mode
<SPIN options>
```

```
  any valid option for spin -a
```

The `ms` option `-dl` causes a model to be contructed in which, for every send posted, Spin will make a nondeterministic choice between forcing the send to be delivered synchronously and allowing it to be buffered. This is necessary for a conservative verification of properties such as freedom from deadlock, but entails a (large) cost in terms of the number of states.

4.2. **Compiling the analyzer.** After executing `ms`, one usually just executes `mscc` with no arguments, though one may add any options that can be added when invoking the C compiler on a standard Spin analyzer (e.g., `-DCOLLAPSE` for better compression).

4.3. **Executing the analyzer.** The successful completion of `mscc` should result in an executable analyzer `pan`. One then executes this in the usual way to perform the verification. Again, one may use any option normally available in Spin, such as `-n`, which suppresses output of unreachable code.

4.4. **Properties.** By default, MPI-Spin checks a number of generic properties that are expected to hold in any model. These include:

(1) the program cannot deadlock (though `-dl` is needed for a full check)
(2) there are never two outstanding requests with buffers that intersect nontrivially
(3) the total number of outstanding requests never exceeds the request bound,
(4) when `MPI_Finalize` is called there are no request objects allocated for and there are no buffered messages destined for the calling process, and
(5) the size of an incoming message is never greater than the size of the receive buffer
(6) `MPI_Init` is called before any other MPI function, `MPI_Init` and `MPI_Finalize` are called at most once (each), and no MPI function other than `MPI_Initialized` or `MPI_Finalized` is called after `MPI_Finalize`.

In addition, program-specific properties may be specified as assertions in the code. Finally, complex temporal properties, including liveness properties, may be specified as never claims, which may be generated using `spin -f` in the usual way.

4.5. **Example.**

```
ex218_mpsc$  ms -DNPRODUCERS=2 -np=3 -buf=1 -req=4 mpsc.prom
MPI-Spin version 1.0 of 20-Sep-2007
ex218_mpsc$ mscc
ex218_mpsc$ ./pan -n
hint: this search is more efficient if pan.c is compiled -DSAFETY
(Spin Version 4.2.9 -- 8 February 2007)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 80 byte, depth reached 2167, errors: 0
    1739 states, stored
    2642 states, matched
    4381 transitions (= stored+matched)
    7705 atomic steps
hash conflicts: 0 (resolved)
```

```
2.827   memory usage (Mbyte)

MPI-Spin memory usage (bytes): 127153
```

4.6. **Examining traces.** If an error is discovered, the trail can be played back in the usual way for doing this with Spin models that use embedded C code. Namely, one executes `./pan -r` possibly with additional arguments to control how much information is included in the trace. Recall that to find a minimal trace with Spin, one can compile the analyzer with `-DREACH` and add the flag `-i` to the invocation of `pan`. This same technique can be used with MPI-Spin.

For example, if we try to perform the same verification of the producer-consumer model but specify a request bound of 3 (instead of 4), an error will be found because there is an execution in which 4 outstanding request can exist at one time. The following edited transcript illustrates how to discover this fact with MPI-Spin:

```
simple$ ms -DNPRODUCERS=2 -np=3 -buf=1 -req=3 mpsc.prom
simple$ mscc -DREACH
simple$ ./pan -i -n
.
.
.
pan: assertion violated MPI: error: request bound exceeded: 3 (at depth 43)
pan: wrote mpsc.prom.trail
(Spin Version 4.2.7 -- 23 June 2006)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 56 byte, depth reached 43, errors: 1
      22 states, stored
       0 states, matched
      22 transitions (= stored+matched)
      22 atomic steps
hash conflicts: 0 (resolved)
.
.
.
pan: wrote mpsc.prom.trail
pan: reducing search depth to 9
.
.
.
State-vector 56 byte, depth reached 43, errors: 18
     101 states, stored
     130 states, matched
     231 transitions (= stored+matched)
```

```
    159 atomic steps
.
.
.
simple$ ./pan -r -n
MPI-Spin initialized.

Rank 2: MPI_Irecv[buffer=0x0,datatype=MPI_POINT,count=0,source=0,tag=0,handle=0]
  Inserting recv request in position 0.

Rank 1: MPI_Isend[buffer=0x0,datatype=MPI_POINT,count=0,dest=2,tag=0,handle=0]

  Inserting send request in position 0.

Rank 2: MPI_Irecv[buffer=0x0,datatype=MPI_POINT,count=0,source=1,tag=0,handle=1]
  Inserting recv request in position 2.

Rank 0: MPI_Isend[buffer=0x0,datatype=MPI_POINT,count=0,dest=2,tag=0,handle=0]
.
.
.
pan: assertion violated MPI: error: request bound exceeded: 3 (at depth 11)
spin: trail ends after 11 steps
```

The trace shows an execution in which first process 2 posts a receive with source 0, then process 1 posts a send, then process 2 posts a receive with source 1, then process 0 posts a send, landing the system in the error state.

5. MPI Primitives Implemented in MPI-Spin

5.1. **General MPI functions.**

5.1.1. `MPI_Init`. Prepare the calling process for MPI communication.

`MPI_Init(proc, rank)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`rank`	a C expression that evaluates to the integer rank for MPI to associate to this process

This function declares that the calling Spin process is an MPI process with the specified rank. It should be called once by each such process, before the process calls any other MPI function. Note that the rank can be different from the Spin PID, although it is often a function of the PID, as in

```
MPI_Init(Pproducer, Pproducer->_pid - 1)
```

You must take care not to assign the same rank to two different processes. (MPI-Spin will complain if you try.) You must also ensure that the set of assigned ranks is $\{0, 1, \ldots, \texttt{NPROCS} - 1\}$.

5.1.2. `MPI_Finalize`. Declare MPI communication to be finished on the calling process.

`MPI_Finalize(proc)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`

This function should be called once by each MPI process. It must be called after `MPI_Init`. No MPI function should be called after `MPI_Finalize`, with the exception of `MPI_Initialized` and `MPI_Finalized`. When `MPI_Finalize` is called, MPI-Spin will check that there are no buffered messages destined for the calling process and that there are no outstanding communication requests for the calling process.

5.1.3. `MPI_Comm_rank`. Get the rank of the calling process.

`MPI_Comm_rank(proc, rank)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`rank`	a C expression evaluating to the address of an integer type variable

This puts the rank of the calling process into the specified variable. It is usually easier to use the C expression macro `RANK(proc)` instead, as in

```
if :: c_expr{ RANK(Pproducer) > 1 } -> ... fi
```

5.1.4. `MPI_Comm_size`. Get the number of processes in the communicator.

`MPI_Comm_size(proc, size)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`size`	a C expression evaluating to the address of an integer variable

Since the number of processes is always specified before executing MPI-Spin, it is usually easier to just use the macro `NPROCS`, as in

```
if :: c_expr{ RANK(Pproducer) == NPROCS - 1 } -> ... fi
```

5.1.5. `MPI_Initialized`. Determine whether this process has been initialized.

`MPI_Initialized(proc)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`

This is a boolean-valued Promela expression which has the value true iff the process has previously called `MPI_Init`. It remains true regardless of whether the process has called `MPI_Finalize`.

5.1.6. `MPI_Finalized`. Determine whether this process has been finalized.

`MPI_Finalized(proc)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`

This is a boolean-valued Promela expression which has the value true iff the process has previously called `MPI_Finalize`.

5.1.7. `MPI_Get_count`. Determine the number of data elements received.

`MPI_Get_count(status, datatype, countPtr)`

`status`	C expression of type `MPI_Status*` pointing to status object
`datatype`	integer type C expression giving the MPI datatype of the elements
`countPtr`	C expression evaluating to address of integer type variable to be filled in

This function is used after a message has been received. The status object that was filled in when the receive operation can then be queried to determine the number of data elements in the message. This value is returned into the variable pointed to by `countPtr`.

5.1.8. `MPI_Pack`. Copy data in inbuf into the outbuf.

`MPI_Pack(inbuf, incount, datatype, outbuf, outsize, position)`

`inbuf`	C expression of type `void*` pointing to inbuf
`incount`	number of elements in inbuf (C expression of integer type)
`datatype`	type of each element
`outbuf`	C expression of type `void*` pointing to outbuf
`outsize`	size (in bytes) of outbuf
`position`	on entry points to part of outbuf for data, incremented on exit

5.1.9. `MPI_Unpack`. Copy data from the inbuf into the outbuf.

`MPI_Unpack(inbuf, insize, position, outbuf, outcount, datatype)`

`inbuf`	C expression of type `void*` pointing to inbuf
`insize`	size (in bytes) of inbuf
`position`	on entry points to data in inbuf to be copied, incremented on exit
`outbuf`	C expression of type `void*` pointing to outbuf
`outcount`	number of elements in outbuf
`datatype`	type of each element

5.2. **MPI blocking point-to-point functions.**

5.2.1. `MPI_Send`. Execute a standard-mode blocking send.

`MPI_Send(proc, buffer, count, datatype, dest, tag)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	C expression of type `void*` pointing to beginning of send buffer
`count`	C expression of integer type evaluating to number of elements in send buffer
`datatype`	one of the MPI datatypes; C expression of integer type
`dest`	the rank of the destination process; C expression of integer type
`tag`	tag to associate to the message; C expression of integer type

All of the integer parameters are unsigned and are limited to the range $[0, 250]$. Predefined datatypes include `MPI_POINT`, `MPI_BYTE`, `MPI_INT`, and `MPI_SYMBOLIC`.

5.2.2. `MPI_Recv`. Execute a blocking receive.

`MPI_Recv(proc, buffer, count, datatype, source, tag, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	C expression of type `void*` pointing to beginning of receive buffer
`count`	C expression of integer type giving upper bound on number of elements to receive
`datatype`	one of the MPI datatypes; C expression of integer type
`source`	the rank of the source process; C expression of integer type
`tag`	the tag of the message to receive; C expression of integer type
`status`	C expression of type `MPI_Status*` pointing to status object

All of the integer parameters are unsigned and are limited to the range $[0, 250]$. Predefined datatypes include `MPI_POINT`, `MPI_BYTE`, `MPI_INT`, and `MPI_SYMBOLIC`. For `source` one may use the special wildcard value `MPI_ANY_SOURCE`. For `tag` one may use the special wildcard value `MPI_ANY_TAG`. The `status` may be `MPI_STATUS_IGNORE` if the status is not needed.

5.2.3. `MPI_Sendrecv`. Execute a blocking send and receive.

```
MPI_Sendrecv(proc, sendbuf, sendcount, sendtype, dest, sendtag,
             recvbuf, recvcount, recvtype, source, recvtag, status)
```

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`sendbuf`	C expression of type `void*` pointing to beginning of send buffer
`sendcount`	C expression of integer type evaluating to number of elements in sendbuf
`sendtype`	datatypes for elements in sendbuf
`dest`	the rank of the destination process; C expression of integer type
`sendtag`	tag to associate to message sent; C expression of integer type
`recvbuf`	C expression of type `void*` pointing to beginning of receive buffer
`recvcount`	C expression of integer type giving upper bound on number of elements to receive
`recvtype`	one of the MPI datatypes; C expression of integer type
`source`	the rank of the source process; C expression of integer type
`recvtag`	the tag of the message to receive; C expression of integer type
`status`	C expression of type `MPI_Status*` pointing to status object

This combines one call to `MPI_Send` and one call to `MPI_Recv` in a single call. The semantics are equivalent to the case where the infrastructure forks off two processes, one to execute the send and one to execute the receive. The function returns only after both operations have completed. It is useful to avoid deadlock in many commonly-occuring scenarios.

5.2.4. `MPI_Sendrecv_replace`. Execute a blocking send and receive with one buffer.

```
MPI_Sendrecv_replace(proc, buffer, count, datatype, dest, sendtag,
                     source, recvtag, status)
```

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	pointer to buffer used for send and receive
`count`	number of elements to be sent and upper bound on number to receive
`datatype`	datatype for send and receive
`dest`	the rank of the destination process; C expression of integer type
`sendtag`	tag to associate to message sent; C expression of integer type
`source`	the rank of the source process; C expression of integer type
`recvtag`	the tag of the message to receive; C expression of integer type
`status`	C expression of type `MPI_Status*` pointing to status object

Like `MPI_Sendrecv`, except a single buffer is used for both the send and receive. The user calls this with the buffer containing the outgoing message. Upon return, the buffer will hold the incoming message. The infrastructure takes care of any temporary buffering required to ensure nothing is overwritten.

5.3. **MPI collective functions.**

5.3.1. `MPI_Barrier`. Block until all procs reach barrier.

```
MPI_Barrier(proc)
```

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`

5.3.2. `MPI_Allreduce`. Apply reduction operation across procs and return result to all.

`MPI_Allreduce(proc, sendbuf, recvbuf, count, datatype, op)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`sendbuf`	pointer to send buffer
`recvbuf`	pointer to receive buffer
`count`	number of elements in send buffer (and receive buffer)
`datatype`	type of each element
`op`	reduction operation, e.g., `MPI_SUM`

Apply reduction operation across all processes to the vectors in the send buffers. The result is returned to all processes in the receive buffers.

5.3.3. `MPI_Reduce`. Apply reduction operation across procs and return result to root.

`MPI_Allreduce(proc, sendbuf, recvbuf, count, datatype, op, root)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`sendbuf`	pointer to send buffer
`recvbuf`	pointer to receive buffer
`count`	number of elements in send buffer (and receive buffer)
`datatype`	type of each element
`op`	reduction operation, e.g., `MPI_SUM`
`root`	integer C expression specifying root

Apply reduction operation across all processes to the vectors in the send buffers. The result is returned to the root's receive buffer. The receive buffer arguments on non-root processes are ignored.

5.3.4. `MPI_Bcast`. Broadcast message to all procs.

`MPI_Bcast(proc, buffer, count, datatype, root)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	pointer to send buffer on root, receive buffer on other procs
`count`	number of elements in buffer (same on all procs)
`datatype`	type of element to send or receive
`root`	integer C expression specifying root

5.4. **MPI nonblocking standard-mode point-to-point functions.**

5.4.1. `MPI_Isend`. Post a non-persistent send request.

`MPI_Isend(proc, buffer, count, datatype, dest, tag, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	C expression of type `void*` pointing to beginning of send buffer
`count`	C expression of integer type evaluating to number of elements in send buffer
`datatype`	one of the MPI datatypes; C expression of integer type
`dest`	the rank of the destination process; C expression of integer type
`tag`	tag to associate to the message; C expression of integer type
`request`	C expression of type `MPI_Request*` giving address of request handle variable

All of the integer parameters are unsigned and are limited to the range $[0, 250]$. Predefined datatypes include `MPI_POINT`, `MPI_BYTE`, `MPI_INT`, and `MPI_SYMBOLIC`.

5.4.2. `MPI_Irecv`. Post a non-persistent receive request.

`MPI_Irecv(proc, buffer, count, datatype, source, tag, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	C expression of type `void*` pointing to beginning of receive buffer
`count`	C expression of integer type giving upper bound on number of elements to receive
`datatype`	one of the MPI datatypes; C expression of integer type
`source`	the rank of the source process; C expression of integer type
`tag`	the tag of the message to receive; C expression of integer type
`request`	C expression of type `MPI_Request*` giving address of request handle variable

All of the integer parameters are unsigned and are limited to the range $[0, 250]$. Predefined datatypes include `MPI_POINT`, `MPI_BYTE`, `MPI_INT`, and `MPI_SYMBOLIC`. For `source` one may use the special wildcard value `MPI_ANY_SOURCE`. For `tag` one may use the special wildcard value `MPI_ANY_TAG`.

5.4.3. `MPI_Wait`. Block till request completes.

`MPI_Wait(proc, request, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`request`	C expression of type `MPI_Request*` giving address of request handle variable
`status`	C expression of type `MPI_Status*` pointing to status object

This statement is enabled iff the request has been completed or successfully canceled. It then fills in the fields of the status object with the appropriate values, and, for a non-persistent request, deallocates the request object and sets the request handle to `MPI_REQUEST_NULL`. For a persistent request, it resets the request object (making it *inactive*) but does not modify the request handle or deallocate the request object.

If the value of the request handle is already `MPI_REQUEST_NULL`, or if the request is inactive, then this function is always enabled and the status will be set to *empty* (the status object with `source` = `MPI_ANY_SOURCE`, `tag` = `MPI_ANY_TAG`, and `count` = 0).

Note: a request that has been successfully canceled is considered to be complete for the purposes of `MPI_Wait` and all other "completion operations."

5.4.4. `MPI_Test`. Determine whether the request has completed.

`MPI_Test(proc, request, flag, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`request`	C expression of type `MPI_Request*` giving address of request handle variable
`flag`	a Promela variable of integer type
`status`	C expression of type `MPI_Status*` pointing to status object

This function is always enabled. It returns 0 into the `flag` variable if request is not complete. If the request has completed, it sets `flag` to 1 and proceeds exactly as in the case of `MPI_Wait` (5.4.3). Note that `flag` is a Promela variable and not a C expression. This allows one to use a Promela variable of type `bit` (or `bool`), which cannot be referred to in a C expression.

Just as in the case of `MPI_Wait`, if the value of the request handle is already `MPI_REQUEST_NULL`, or if the request is inactive, then this will set `flag` to 1 and the status will be set to *empty*.

Note: a request that has been successfully canceled is considered to be complete for the purposes of `MPI_Test` and all other completion operations. In this case `flag` will be set to 1 and the status object to *canceled*.

5.4.5. `MPI_Request_free`. Deallocate request object.

`MPI_Request_free(proc, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`request`	C expression of type `MPI_Request*` giving address of request handle variable

If the request has completed, been successfully canceled, or is inactive, this operation deallocates the request object. Otherwise, the request object is effectively marked to be deallocated as soon as it completes, is successfully canceled, or becomes inactive. In either case, the request handle variable is immediately set to `MPI_REQUEST_NULL`.

Note: the request may *not* be `MPI_REQUEST_NULL`, and MPI-Spin will complain (with an assertion violation) if it is.

5.4.6. `MPI_Request_get_status`. Get the status of a request without destroying it.

`MPI_Request_get_status(proc, request, flag, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`request`	C expression of type `MPI_Request*` giving address of request handle variable
`flag`	a Promela variable of integer type (usually `bit` or `bool`)
`status`	C expression of type `MPI_Status*` pointing to status object

If the request has completed, this function fills in the status fields and sets `flag` to 1. The request is not deallocated and the request handle is not modified. If the request has not completed (or is inactive), it sets `flag` to 0 and `status` to *undefined*.

An assertion violation occurs if the request is `MPI_REQUEST_NULL`.

The special constant `MPI_STATUS_IGNORE` can be used for the `status` argument if the status is not needed.

If the request has been successfully canceled then this will set `flag` to 1 and the `status` fields will be set to certain value which indicate a *canceled* request. A subsequent call to `MPI_Test_cancelled` (§5.4.16) on the status object can be used to determine whether the request was canceled.

5.4.7. `MPI_Waitany`. Wait for at least one request in an array to complete then process it.

`MPI_Waitany(proc, count, requestArray, index, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`count`	C expression of integer type giving number of requests in array
`requestArray`	array of requests of length count; C expression of type `MPI_Request*`
`index`	pointer to a byte variable which will be set; C expression of type `uchar*`
`status`	C expression of type `MPI_Status*` pointing to status object

This function blocks until at least one of the active requests has completed, then chooses one such request, sets `index` to the index of the chosen request in the request array, and proceeds as in the case of `MPI_Wait` (§5.4.3) with the chosen request.

A request that has been successfully canceled is considered to be complete for these purposes.

Some or all of the requests may be inactive or `MPI_REQUEST_NULL`. If all requests are inactive or `MPI_REQUEST_NULL` then this function returns without blocking with `status` set to *empty* and `index` set to `MPI_UNDEFINED`.

5.4.8. `MPI_Testany`. Determine if a request in an array has completed and if so process it.

`MPI_Testany(proc, count, requestArray, index, flag, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`count`	C expression of integer type giving number of requests in array
`requestArray`	array of requests of length count; C expression of type `MPI_Request*`
`index`	pointer to a byte variable which will be set; C expression of type `uchar*`
`flag`	a Promela variable of integer type (usually `bit` or `bool`)
`status`	C expression of type `MPI_Status*` pointing to status object

Determines without blocking whether at least one of the given active requests has completed. If so, one such request is selected, `flag` is set to 1, `index` is set to the index in the request array of the chosen request, and the request is processed in the manner of `MPI_Wait` (§5.4.3). If none of the requests has completed, `flag` is set to 0, `index` to `MPI_UNDEFINED`, and `status` to *undefined*.

If all requests are inactive or `MPI_REQUEST_NULL`, `flag` is set to 1, `index` to `MPI_UNDEFINED`, and `status` to *empty*.

A request that has been successfully canceled is considered to be complete for these purposes.

5.4.9. `MPI_Waitall`. Wait for all requests in an array to complete, then process them.

`MPI_Waitall(proc, count, requestArray, statusArray)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`count`	C expression of integer type giving number of requests in array
`requestArray`	array of requests of length count; C expression of type `MPI_Request*`
`statusArray`	array of status objects; C expression of type `MPI_Status*`

This function blocks until all active requests have completed, then sets all the corresponding statuses in the status array. It has the same effect as executing `MPI_Wait` on each request/status in any order.

If there's no need to know the statuses, the constant `MPI_STATUSES_IGNORE` can be used for the `statusArray` argument.

A request that has been successfully canceled is considered complete for these purposes.

Some or all of the requests may be inactive or `MPI_REQUEST_NULL`. This function is enabled as soon as all active requests have completed. All statuses associated to inactive or null requests are set to *empty*.

5.4.10. `MPI_Testall`. Determine if all requests in an array have completed and if so process them.

`MPI_Testall(proc, count, requestArray, flag, statusArray)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`count`	C expression of integer type giving number of requests in array
`requestArray`	array of requests of length count; C expression of type `MPI_Request*`
`flag`	a Promela variable of integer type (usually `bit` or `bool`)
`statusArray`	array of status objects; C expression of type `MPI_Status*`

This function determines without blocking whether all active requests in the list have completed. If they have, it sets `flag` to 1 and fills in the status array with the appropriate values. Otherwise, it sets `flag` to 0.

A request that has been successfully canceled is considered complete for these purposes.

Some or all of the requests may be inactive or `MPI_REQUEST_NULL`. This functions sets `flag` to 1 if all active requests have completed. All statuses associated to inactive or null requests are set to *empty*.

5.4.11. `MPI_Waitsome`. Wait until at least one request completes then process the completed ones.

`MPI_Waitsome(proc, incount, requestArray, outcount, indexArray, statusArray)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`incount`	the length of `requestArray`; C expression of integer type
`requestArray`	array of requests of length `incount`; C expression of type `MPI_Request*`
`outcount`	location for number of completed requests; C expression of type `uchar*`
`indexArray`	location for indices of completed requests; C expression of type `uchar*`
`statusArray`	array of status objects; C expression of type `MPI_Status*`

This function becomes enabled what at least one of the active requests in the list has completed. It then processes all those requests that have completed. It returns, in `outcount`, the number of completed requests, and it fills in the first `outcount` entries in `indexArray` with the indices of the completed requests. It also fills in the first `outcount` entries of `statusArray` with the statuses of the completed requests.

If all the requests are inactive or `MPI_REQUEST_NULL`, this returns immediately with `outcount` set to `MPI_UNDEFINED`.

The constant `MPI_STATUSES_IGNORE` can be used if the statuses are not needed.

5.4.12. `MPI_Testsome`. Process all completed requests in an array.

`MPI_Testsome(proc, incount, requestArray, outcount, indexArray, statusArray)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`incount`	the length of `requestArray`; C expression of integer type
`requestArray`	array of requests of length `incount`; C expression of type `MPI_Request*`
`outcount`	location for number of completed requests; C expression of type `uchar*`
`indexArray`	location for indices of completed requests; C expression of type `uchar*`
`statusArray`	array of status objects; C expression of type `MPI_Status*`

If at least one of the active requests in the list has completed, this returns in `outcount` the number of such requests and the corresponding information in `indexArray` and `statusArray`. If all the requests are inactive or `MPI_REQUEST_NULL`, this sets `outcount` to `MPI_UNDEFINED`. If there is at least one active request but none of the active requests has completed, this sets `outcount` to 0.

The constant `MPI_STATUSES_IGNORE` can be used if the statuses are not needed.

5.4.13. `MPI_Iprobe`. Determines if there is an incoming message matching the given parameters.

`MPI_Iprobe(proc, source, tag, flag, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`source`	rank of source (integer C expr) or `MPI_ANY_SOURCE`
`tag`	message tag (integer C expr) or `MPI_ANY_TAG`
`flag`	Promela variable of integer type (usually boolean)
`status`	pointer to status object (C expr of type `MPI_Status*`)

Sets `flag` to 1 if there is a visible message or send request destined for this process whose source and tag parameters match those specified in the arguments to this function. Otherwise sets `flag` to 0. It does not block. If `flag` is set to 1 then the status object is also filled in.

5.4.14. `MPI_Probe`. Block until there is an incoming message matching the given parameters.

`MPI_Probe(proc, source, tag, flag, status)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`source`	rank of source (integer C expr) or `MPI_ANY_SOURCE`
`tag`	message tag (integer C expr) or `MPI_ANY_TAG`
`status`	pointer to status object (C expr of type `MPI_Status*`)

Like `MPI_Iprobe`, only this one blocks until a matching message or request is detected, rather than setting a flag.

5.4.15. `MPI_Cancel`. Attempt to cancel a communication request.

`MPI_Cancel(proc, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`request`	pointer to request handle variable (C expr of type `MPI_Request*`

This attemtps to cancel the request associated with `request`. If the request has already completed or been canceled, this is effectively a no-op. A completion operation (e.g., `MPI_Wait`) still needs to be invoked on the request. The status object can then be queried to determine whether the communication was canceled. Note that it is possible to cancel a send request after it becomes visible. Hence it is possible to cancel a send request after a probe on the receiver side has successfully probed the request.

5.4.16. `MPI_Test_cancelled`. Examine status to determine if request was really canceled.

`MPI_Test_cancelled(status, flag)`

`status`	pointer to status object (C expr of type `MPI_Status*`)
`flag`	Promela variable of integer type (usually boolean)

This determines whether the communication that resulted in the given status completed normally or was canceled. If it was canceled, the `flag` is set to 1, else it is set to 0.

5.4.17. `MPI_Send_init`. Create an inactive persistent send request.

`MPI_Send_init(proc, buffer, count, datatype, dest, tag, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	C expression of type `void*` pointing to beginning of send buffer
`count`	C expression of integer type evaluating to number of elements in send buffer
`datatype`	one of the MPI datatypes; C expression of integer type
`dest`	the rank of the destination process; C expression of integer type
`tag`	tag to associate to the message; C expression of integer type
`request`	C expression of type `MPI_Request*` giving address of request handle variable

This initiates a persistent send request. The parameters have the same meaning as the ones for `MPI_Isend` (§5.4.1). However, in the case of a persistent request, the new request is *inactive*, i.e., is not associated with any ongoing communication operation. To activate the request, one must invoke `MPI_Start`. Also, after calling one of the completion operations (e.g., `MPI_Wait`), the persistent request is *not* deallocated. Instead, it is returned to the inactive state, so it can be started again. To deallocate a persistent request, one must use `MPI_Request_free` (§5.4.5).

5.4.18. `MPI_Recv_init`. Create an inactive persistent receive request.

`MPI_Recv_init(proc, buffer, count, datatype, source, tag, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`buffer`	C expression of type `void*` pointing to beginning of receive buffer
`count`	C expression of integer type giving upper bound on number of elements to receive
`datatype`	one of the MPI datatypes; C expression of integer type
`source`	the rank of the source process; C expression of integer type
`tag`	the tag of the message to receive; C expression of integer type
`request`	C expression of type `MPI_Request*` giving address of request handle variable

This creates an inactive persistent receive request. See the notes on persistent requests in §5.4.17.

5.4.19. `MPI_Start`. Make an inactive persistent request active.

`MPI_Start(proc, request)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`request`	C expression of type `MPI_Request*` giving address of request handle variable

Starts an inactive persistent request. If the request is already active, an error is reported.

5.4.20. `MPI_Startall`. Make all requests in an array of inactive requests active.

`MPI_Startall(proc, count, requestArray)`

`proc`	the letter `P` followed by the proctype name, e.g. `Pproducer`
`count`	number of requests in array (C expr of integer type)
`requestArray`	array of requests of length `count`; C expression of type `MPI_Request*`

Applies `MPI_Start` to each of the requests in the array.

5.5. **Types.**

(1) `MPI_Request`: a handle for a request object
(2) `MPI_Status`: an object with fields for source, tag, count, "cancelation", error
(3) `MPI_Symbolic`: a symbolic arithmetic expression
(4) `MPI_Op`: a reduction operation, e.g. `MPI_SUM`

5.6. **Constants.**

(1) `MPI_ANY_SOURCE`: used to receive messages from any source
(2) `MPI_ANY_TAG`: used to receive messages with any tag
(3) `MPI_STATUS_IGNORE`: used for status argument when you don't care about status
(4) `MPI_STATUSES_IGNORE`: like above but in place of an array of statuses
(5) `MPI_REQUEST_NULL`: a handle that doesn't refer to any request object
(6) `MPI_BYTE`: an MPI datatype corresponding to Promela's `byte`, C's `unsigned char`
(7) `MPI_SHORT`: an MPI datatype corresponding to Promela/C's `short` (2 bytes)
(8) `MPI_INT`: an MPI datatype corresponding to Promela/C's `int` (4 bytes)
(9) `MPI_POINT`: a virtual datatype that takes up 0 bytes
(10) `MPI_SYMBOLIC`: a new datatype corresponding to `MPI_Symbolic` (4 bytes)
(11) `MPI_UNDEFINED`: a constant used to represent an undefined value
(12) `MPI_OP_NULL`: undefined reduction operation
(13) `MPI_MAX`: maximum reduction operation
(14) `MPI_MIN`: minimum reduction operation
(15) `MPI_SUM`: sum reduction operation
(16) `MPI_PROD`: product reduction operation
(17) `MPI_LAND`: logical and reduction operation
(18) `MPI_BAND`: bit-wise and reduction operation
(19) `MPI_LOR`: logical or reduction operation
(20) `MPI_BOR`: bit-wise or reduction operation
(21) `MPI_LXOR`: logical xor reduction operation
(22) `MPI_BXOR`: bit-wise xor reduction operation
(23) `MPI_MAXLOC`: max value and location reduction operation
(24) `MPI_MINLOC`: min value and location reduction operation

6. Using Symbolic Algebra in MPI-Spin

Symbolic expressions can be incorporated into an MPI-Spin model using the type `MPI_Symbolic` together with a number of constants and functions for that type. Integer, real, and boolean values can all be represented symbolically.

6.1. **Creating symbolic expressions.** Integer and boolean concrete values can be transformed into symbolic values. For example, the following constants of type `MPI_Symbolic` are always available in either C or Promela mode:

- `SYM_ZERO`: zero (0)
- `SYM_ONE`: one (1)
- `SYM_FALSE`: the boolean value `false`
- `SYM_TRUE`: the boolean value `true`

Moreover, the following C function returns a symbolic expression for any given integer:

6.1.1. `SYM_intConstant`. Return an integer as a symbolic expression.

```
SYM_intConstant(int n)
  n  any integer
```

Note that this function, and in fact all the functions that operate on symbolic expressions, can only be used in C mode.

A symbolic constant can be thought of as a symbol such as x_i. Such an expression is returned by the following function:

6.1.2. `SYM_symbolicConstant`. Return the symbolic constant x_i.

```
SYM_symbolicConstant(int i)
  i  nonnegative integer index for the symbolic constant
```

Conceptually, the symbolic constants $x_0, x_1, \ldots$ exist and this function just returns one of them. In other words, it is perfectly fine to call this function twice with the same value for i; the function will just return the same symbolic constant x_i each time.

6.2. **Operations on symbolic expressions.** New symbolic expressions can be formed from old ones by applying the functions described here. When applied to real-valued symbolic expressions, the exact behavior of these operations depends on the value of the *arithmetic mode* when the operation is called. There are three modes, numbered 0, 1, and 2: (0) Herbrand, (1) IEEE, and (2) Real.

In Herbrand mode (0), all operations are treated as uninterpreted functions. Hence no simplifications or reorderings are performed at all.

In IEEE mode (1), certain manipulations are performed that are guaranteed to preserve the evaluation of the expression in any IEEE754-compliant arithmetic. Roughly speaking, this IEEE standard specifies that the result of floating-point addition, multiplication, subtraction, division, and negation must be the same as the result obtained by first performing the operation exactly and then rounding (using one of several specified rounding functions). Some consequences of this are the following: (1) addition and multiplication are commutative, (2) $1x = x = x1$, $0 + x = x = x + 0$, and $x - x = 0$ for any floating-point value x, and (3) $x/x = 1$ for non-zero x. In IEEE mode, the operations below take advantage of these (and other similar) facts to simplify the expressions returned whenever possible.

In Real mode (2), manipulations are performed that are guaranteed to preserve the evaluation of the expression when treated as an expression of real numbers, i.e., with full precision arithmetic.

The following functions take numeric (real- or integer-valued) symbolic expressions as arguments and return a numeric symbolic expression:

```
MPI_Symbolic SYM_add(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_subtract(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_multiply(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_divide(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_sqrt(MPI_Symbolic operand);  /* square root */
MPI_Symbolic SYM_abs(MPI_Symbolic operand); /* absolute value */
```

The following operations return a boolean-valued symbolic expression. The paramters `p` and `q` represent boolean-valued symbolic expressions, while `x` and `y` represent numeric symbolic expressions.

```
MPI_Symbolic SYM_equals(MPI_Symbolic x, MPI_Symbolic y); /* x == y */
MPI_Symbolic SYM_nequals(MPI_Symbolic x, MPI_Symbolic y); /* x != y */
MPI_Symbolic SYM_lessThan(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_greaterThan(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_lessThanOrEquals(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_greaterThanOrEquals(MPI_Symbolic x, MPI_Symbolic y);
MPI_Symbolic SYM_conjunct(MPI_Symbolic p, MPI_Symbolic q); /* p && q */
MPI_Symbolic SYM_negate(MPI_Symbolic p); /* !p */
```

6.3. **Reasoning about symbolic expressions.**

6.3.1. `SYM_isEqualTo`. Determine whether condition implies equality.

```
int SYM_isEqualTo(MPI_Symbolic pc, MPI_Symbolic x, MPI_Symbolic y)
```

`pc`	a boolean-value symbolic expression
`x`	any symbolic expression
`y`	any symbolic expression

Let p_1 denote the proposition $\texttt{pc} \Rightarrow x = y$ and let p_0 denote the proposition $\texttt{pc} \Rightarrow x \neq y$. This function returns one of three values: 1, 0, or -1. A value of 1 means that p_1 is a theorem. A value of 0 means that p_0 is a theorem. A value of -1 means "don't know," i.e., the (simple) theorem-proving techniques used to implement this function could prove neither p_0 nor p_1. Note that a returned value of -1 may be returned either because neither p_0 nor p_1 is a theorem (as is the case, for example, if $\texttt{pc} \equiv \texttt{true}$, and x and y are distinct symbolic constants), or because one of them is a theorem but the theorem-proving techniques were too weak to discover this fact.

The symbol `pc` is used because it stands for *path condition*. The path condition, a concept from symbolic execution, represents the condition that must have held in order for execution to have followed the path that it did. The most common application of this function deals with modeling conditional branches in models that keep track of the path condition. Typically, if the original code contains a branch of the type

```
if (x==y) {...} else {...}
```

then the model would begin by invoking this function. If the value returned is 1 or 0 then the appropriate branch is taken. If the value returned is -1 then the model must make a nondeterministic choice between 0 and 1, update the path condition appropriately, and then take the branch. During verification, all possible nondeterministic choices will be explored and hence all possible executions of the program will be explored (and possibly some infeasible executions as well). For details on this approach, see [6].

6.3.2. `SYM_isLessThan`. Determine whether condition implies $x < y$.

```
int SYM_isLessThan(MPI_Symbolic pc, MPI_Symbolic x, MPI_Symbolic y)
```

`pc`	a boolean-value symbolic expression
`x`	any symbolic expression
`y`	any symbolic expression

Like above, but with p_1 denoting $\texttt{pc} \Rightarrow x < y$ and p_0 denoting $\texttt{pc} \Rightarrow x \geq y$.

7. Examples

A number of examples can be found in the `examples` directory of MPI-Spin. Most of them can be executed by simply typing `make`. The best way to learn to use MPI-Spin is to examine the source and make files for these examples.

7.1. Examples from *MPI—The Complete Reference.*

The subdirectory of `examples` called `MPI-TheCompleteReference` contains examples from the popular MPI text [7]. Many correctness properties were verified for these examples, and, in two cases, faults were disovered. The following is a summary of these results.

Example 2.17: *Use of nonblocking communications in Jacobi computation.* There is a fault in the code that is revealed by certain configurations. The problem occurs when on at least one process, $m = 1$, which will happen whenever $n < 2p$. Then two sends are posted from the same buffer (the single column of local matrix B on at least one process). For all configurations we examined outside of this problematic region, we are able to verify that the sequential and parallel versions are Herbrand equivalent, using the symbolic execution method. (The sequential version is given as Example 2.12 earlier in the book.)

Example 2.18: *Multiple producer, single-consumer code using nonblocking communication.* We are able to verify four properties:

p_0: freedom from deadlock and standard assertions,
p_1: every message produced is eventually consumed,
p_2: no producer becomes permanently blocked,
p_3: for a fixed producer, messages are consumed in the order produced.

Example 2.19: *Multiple producer, single-consumer code, modified to use test calls.* There is a bug. The code

```
i = 0;
while (1) { /* main loop */
  for (flag=0; !flag; i=(i+1)%(size-1)) {
    MPI_Test(&(buffer[i].req), &flag, &status);
  }
  consume_and_post(i);
}
```

(where *consume_and_post(i)* stands in for code that consumes from `buffer[i]` and then posts another receive request to process i) is incorrect: the statement `i=(i+1)%(size-1)` is erroneously executed after the call to `MPI_Test` sets `flag` to true but before exiting the loop and calling *consume_and_post(i)*. The correct code should look something like the following:

```
i = 0;
while (1) { /* main loop */
  flag = 0;
  while (1) {
    MPI_Test(&(buffer[i].req), &flag, &status);
    if (flag) break;
    i=(i+1)%(size-1);
  }
  consume_and_post(i);
  i=(i+1)%(size-1);
}
```

The fault was revealed in attempting to verify p_0 in a small configuration. The failure reported by MPI-Spin was an attempt to exceed the specified bound on the number of outstanding requests. The configuration consisted of 3 processes, no buffering, and an upper bound of 4 outstanding requests. After correcting the bug, we verified all 4 properties over a range of configurations.

Example 2.20: *An example using MPI_REQUEST_FREE.* We verified freedom from deadlock and standard assertions.

Example 2.21: *Message ordering for nonblocking operations.* Verified freedom from deadlock, standard assertions, and that the messages were always matched with the correct receives.

Example 2.22: *Order of completion for nonblocking operations.* Freedom from deadlock, standard assertions, and that in either process, the two requests could complete in either order.

Example 2.23: *An illustration of progress semantics.* Freedom from deadlock, standard assertions, and the correct values are received by both receives.

Example 2.24: *An illustration of buffering for nonblocking messages.* Freedom from deadlock, standard assertions, and the correct values are received by both receives.

Example 2.25: *Out of order communication with nonblocking messages.* The code in the book clearly does not correspond to the discussion given in the text. I took the code from the first edition instead. Freedom from deadlock, standard assertions, and the correct values are received by both receives.

Example 2.26: *Producer-consumer code using waitany.* This is the third version of the multiple-producer single-consumer example to be considered in the book. It replaces the busy-wait test loop in Example 2.19 with a single call to `MPI_Waitany`. As pointed out in the book, this has the disadvantage that it allows "starvation"—the consumer can repeatedly consume messages from one process while ignoring messages sent by all other procesess. This is reflected in the analysis:

p_0: still holds

p_1: fails. Spin finds a counterexample which ends up in an infinite loop in which producer 1 has sent a message and the request has completed, but the consumer repeatedly chooses the message from process 0 in the `MPI_Waitany`.

p_2: also fails, even with progress assumptions and weak fairness. Spin produces a counterexample in which producer 1 sends a message, this is matched by the receive posted by the consumer, completes, producer 1 posts the second send, this one isn't matched (because the `MPI_Waitany` in the consumer repeatedly returns the producer 0 match), and it doesn't complete (the MPI infrastructure has decided to block this completion until a matching receive is posted), so the producer blocks at the wait. Meanwhile the consumer repeatedly selects the match for producer 0.

p_3: still holds.

Example 2.27: *Main loop of Jacobi computation using waitall.* Exact same results as for Example 2.17.

Example 2.28: *A client-server code where starvation is prevented.* (Note *client-server* should be *producer-consumer.*) Fourth version of the multiple-producer single-consumer example. It replaces the `MPI_Waitany` of Example 2.26 with an `MPI_Waitsome`. This is to correct the starvation problem. All 4 properties hold.

Example 2.29: *Use a blocking probe to wait for an incoming message.* There's an `i` used for two different variables but otherwise the program appears to be correct. We verified freedom from deadlock and the standard assertions and that the values are received into `i` and `x`.

Example 2.30: *A similar program to the previous example, but with a problem.* An incorrect use of probe, and, sure enough, Spin catches the bug.

Example 2.31: *Code using MPI_CANCEL.* I had to add a line to wait for the completion of the request in process 1 after posting the second receive. Without that, you can terminate with an unfreed request object still floating around. Otherwise everything seems to be OK.

Example 2.32: *Jacobi computation, using persistent requests.* Everything works very well (modulo the usual error).

Example 2.33: *Starvation-free producer-consumer code.* This is the 5th and final version of the multiple-producer, single-consumer system. Like the third version, Example 2.26, this uses an `MPI_Waitany` at each loop iteration to select and process one completed receive request. However, unlike that version, starvation is prevented by a judicious use of `MPI_REQUEST_NULL`.

References

[1] Gerard J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[2] Gerard J. Holzmann. *The Spin Model Checker.* Addison-Wesley, Boston, 2004.

[3] Message Passing Interface Forum. MPI: A Message-Passing Interface standard, version 1.1. `http://www.mpi-forum.org/docs/`, 1995.

[4] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. `http://www.mpi-forum.org/docs/`, 1997.

[5] Stephen F. Siegel. Model checking nonblocking MPI programs. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007, Proceedings*, LNCS, 2007. To appear.

[6] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori Pollock and Mauro Pezzé, editors, *ISSTA 2006: Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis*, pages 157–168, Portland, ME, 2006.

[7] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference, Volume 1: The MPI Core.* MIT Press, second edition, 1998.

Dept. of Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA
E-mail address: `siegel@cis.udel.edu`