Verifying Parallel Programs with MPI-Spin
Part 2: Language

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

EuroPVM/MPI 2007
Paris, France
30 September 2007

Overview

1. High-level structure
2. Processes
3. Variables and Types
4. Statements
5. Misc.
   - inlines
   - pre-processor
6. Example: Diffusion model

High-level structure

At the highest level, a model is a sequence of the following elements:

1. process type definitions
2. user-defined type declarations
3. global variable declarations
   - variables shared by all processes
4. inlines
5. embedded C code

Process type definitions

- defines a process type (code and local state)
- but does not instantiate any processes of that type
- processes are instantiated and run explicitly using `run`
- more flexible than typical MPI style

Process type definitions

- defines a process type (code and local state)
- but does not instantiate any processes of that type
- processes are instantiated and run explicitly using `run`
- more flexible than typical MPI style
  - MPI style
    - `if (myrank == 0) { master(); } else { slave(); }`
  - MPI-Spin style
    - one proctype for master, another for slave

Process type definitions

- defines a process type (code and local state)
- but does not instantiate any processes of that type
- processes are instantiated and run explicitly using `run`
- more flexible than typical MPI style
  - MPI style
    - `if (myrank == 0) { master(); } else { slave(); }`
  - MPI-Spin style
    - one proctype for master, another for slave
- syntax

```
BEGIN_MPI_PROC(procname)

  : (process body: local decls, statements)

END_MPI_PROC(procname)
```

Process instantiation

- `run procname()`
- can instantiate multiple processes of a single type
- maximum of 255 active processes at any time

Process instantiation

- `run procname()`
- can instantiate multiple processes of a single type
- maximum of 255 active processes at any time
- each process has a unique SPIN pid between 0 and 255
  - assigned in order of instantiation
  - stored in local variable `_pid`
  - SPIN pid may or may not be the same as the MPI rank

Process instantiation

- `run procname()`
- can instantiate multiple processes of a single type
- maximum of 255 active processes at any time
- each process has a unique SPIN pid between 0 and 255
  - assigned in order of instantiation
  - stored in local variable `_pid`
  - SPIN pid may or may not be the same as the MPI rank
- the `init` process
  - a special process instantiated automatically
  - syntax: `init { ... }`
  - it is optional

Active process type definition

- just like regular process type definition
  - but takes additional integer literal parameter
  - specifies number of processes to instantiate immediately
  - these processes will exist and be running in the initial state

Active process type definition

- just like regular process type definition
  - but takes additional integer literal parameter
  - specifies number of processes to instantiate immediately
  - these processes will exist and be running in the initial state

```
BEGIN_ACTIVE_MPI_PROC(procname, number)
```

- syntax ⋮

```
END_ACTIVE_MPI_PROC(procname)
```

Active process type definition

- just like regular process type definition
  - but takes additional integer literal parameter
  - specifies number of processes to instantiate immediately
  - these processes will exist and be running in the initial state

```
BEGIN_ACTIVE_MPI_PROC(procname, number)
```

- syntax

```
⋮
END_ACTIVE_MPI_PROC(procname)
```

```
BEGIN_ACTIVE_MPI_PROC(proc, NPROCS)
```

- typical usage

```
⋮
END_ACTIVE_MPI_PROC(proc)
```

  - `NPROCS` is a macro set to the number of processes
  - value specified on command line: `ms ... -np=5 ...`

The MPI dæmon process

- all MPI-Spin models should include the MPI dæmon process
- models all aspects of the MPI infrastructure
  - matches send and receive requests
  - uploads (buffers) messages from send buffer to system buffer
  - downloads messages from system buffer to receive buffer
  - etc.
- can be included in the active or inactive style
  - active
    - insert `ACTIVE_MPI_DAEMON` somewhere at top level
  - inactive
    - insert `MPI_DAEMON` somewhere at top level
    - start dæmon with `RUN_MPI_DAEMON`

Variables

- there are only 2 scopes

Variables

- there are only 2 scopes
  1. global
     - i.e., shared by all processes
     - declared at highest level

Variables

- there are only 2 scopes
  1. global
     - i.e., shared by all processes
     - declared at highest level
  2. local
     - i.e., local to a single process
     - declared in proctype declaration
     - local variables may be declared anywhere in the proctype body
     - safer practice: put all such decls at beginning of body
     - SPIN moves them all there anyway

Variables

- there are only 2 scopes
  1. global
     - i.e., shared by all processes
     - declared at highest level
  2. local
     - i.e., local to a single process
     - declared in proctype declaration
     - local variables may be declared anywhere in the proctype body
     - safer practice: put all such decls at beginning of body
     - SPIN moves them all there anyway
- declaration syntax is C-like:
  - `byte a`
  - `int x[10]`
  - etc.

Types

1. integer types
2. arrays
3. user-defined types
4. MPI types
   - `MPI_Request`
   - `MPI_Status`
   - `MPI_Symbolic`

Integer types

- `bit` or `bool`
  - 0=`false`, 1=`true`
  - no C equivalent
- `byte`
  - unsigned: 0..255
  - corresponds to C `unsigned char`
- `short`
  - signed, corresponds to C `short int`
  - typically 2 bytes, range $-2^{15}..2^{15}-1$
- `int`
  - signed, corresponds to C `int`
  - typically 4 bytes, range $-2^{31}..2^{31}-1$

Arrays

- 1-dimensional arrays of static extent are supported
  - `int x[10] = 5`
    - an array of 10 ints
    - all entries initialized to 5

Arrays

- 1-dimensional arrays of static extent are supported
  - `int x[10] = 5`
    - an array of 10 ints
    - all entries initialized to 5
- if you need a 2-d (or higher dimensional) array
  - this can be approximated by combining 1-d arrays with user-defined types...

User-defined types

- similar to C's "structs"

```
typedef Field {
  short f = 3;
  byte g
};
typedef Record {
  byte a[3];
  int fld1;
  Field fld2;
  bit b
};
Record rec;

    ⋮

  rec.fld2.g = 255;
```

Higher-dimensional arrays

- can be represented as an array of a user-defined type with a field that is an array ...

```
typedef Row {
  byte data[NUM_COLS]
};

Row matrix[NUM_ROWS];
.
.
.
    matrix[i].data[j] = 255;
.
.
.
```

MPI types

- `MPI_Request`
  - used exactly the same as in MPI
  - a value of type `MPI_Request` is a request handle
  - a request handle is a reference to a request object
  - request objects are created by calls to `MPI_Isend`, `MPI_Irecv`, ...
- `MPI_Status`
  - just like in MPI
  - a value of type `MPI_Status` is a status object
  - it is a structure with at least the following fields
    - `source`
    - `tag`
- `MPI_Symbolic`
  - used to represent a numerical value symbolically
  - e.g., "`x0+2*(x1*x1+x2*x2)/x7`"

Statements

1. assignment
2. expression
3. selection
4. loop
5. `printf`
6. `run`
7. `assert`
8. `atomic` and `d_step`
9. `c_code` and `c_expr`
10. MPI functions

assignment

- syntax
  - `var = expr`

assignment

- syntax
  - `var = expr`
- guard: `true`
  - i.e., assignments are always enabled

assignment

- syntax
  - `var = expr`
- guard: `true`
  - i.e., assignments are always enabled
- `x++`
  - syntactic sugar for `x=x+1`
- `x--`
  - syntactic sugar for `x=x-1`

expression

- syntax
  - `expr`
  - i.e., any expression can also be used as a statement

expression

- syntax
  - `expr`
  - i.e., any expression can also be used as a statement
- guard: `expr`
  - i.e., the statement blocks iff `expr` evaluates to `false`
  - if `expr` evaluates to `true`, it executes a no-op
    - assuming `expr` is side-effect-free

expression

- syntax
  - `expr`
  - i.e., any expression can also be used as a statement
- guard: `expr`
  - i.e., the statement blocks iff `expr` evaluates to `false`
  - if `expr` evaluates to `true`, it executes a no-op
    - assuming `expr` is side-effect-free
- example:
  - `x = 1; x; ...`
  - will this block at statement `x`?

expression

- syntax
  - `expr`
  - i.e., any expression can also be used as a statement
- guard: `expr`
  - i.e., the statement blocks iff `expr` evaluates to `false`
  - if `expr` evaluates to `true`, it executes a no-op
    - assuming `expr` is side-effect-free
- example:
  - `x = 1; x; ...`
  - will this block at statement `x`?
  - maybe, maybe not: another process may set `x` to 0

Expressions

- integer operations
  - `+`, `-`, `*`, `/`, `%`
- comparisons (yield boolean value)
  - `<`, `>`, `<=`, `>=`, `!=`, `==`
- boolean operations
  - `&&`, `||`, `!`

selection: syntax

- syntax:

```
if
:: p1 -> s11; s12; s13; ...
:: p2 -> s21; s22; s23; ...
                .
                .
                .
:: pn -> sn1; sn2; sn3; ...
fi
```

- each clause consists of a guard followed by a sequence of statements

selection: semantics

- the statement is enabled if at least one guard evaluates to `true`
- execution consists of selecting one clause with an enabled guard and shifting program counter to point just after guard
- after last statement executes, program counter is moved to point just after `fi`
- note: other processes can execute between statements, and between guard and first statement
- guard for whole statement: `p1 || p2 || ... || pn`
- `else`
  - special guard
  - only selected if all other guards are false
  - guarantees that the whole statement will never block

loop

- syntax:

```
do
:: p1 -> s11; s12; s13; ...
:: p2 -> s21; s22; s23; ...
                .
                .
                .
:: pn -> sn1; sn2; sn3; ...
od
```

- semantics
  - like `if`, but after last statement in sequence executes, program counter returns to point just before `do`
  - the `break` statement is the only way to break out of a loop

printf

- similar to C
- example

```
printf("numbers: %d\t%d\n", i, x[2*i+j])
```

assert

- example
  ```
  assert(x==5 && y<z)
  ```
- causes error to be reported if assertion fails
- in verification mode, SPIN checks that assertions can never be violated

atomic and d_step

- atomic
  - a sequence of statements can be placed within

    atomic { ... }
  - no other processes will be scheduled while inside atomic
    - exception: if a statement inside the atomic blocks the process loses atomicity and another process can be scheduled
    - if at some future point the first process gets scheduled then it regains atomicity
  - guard: guard of first statement in sequence
- d_step
  - like atomic, but even more so
    - no statement inside the d_step can block
    - no nondeterministic choice can occur inside the d_step
    - only one entry point "{" and one exit point "}"
    - defines a single atomic transition

c_code

- a single atomic transition can be described using arbitrary C code in a `c_code { ... }` statement
  - C code is treated like a "black box" by SPIN
  - pointers, arrays, functions, ... are all allowed
  - no nondeterministic choice or blocking allowed in C code
  - refer to global Promela variables by pre-appending `now.`
  - refer to local Promela variables by pre-appending `Pprocname->`
- example

```
int x;
BEGIN_ACTIVE_MPI_PROC(proc, 2)
  int a[10];
  c_code {
    int i;
    for (i = 0; i < 10; i++) Pproc->a[i] = i*i*now.x;
  }
                  ⋮
```

c_expr

- any side-effect-free C expression can be placed inside

  `c_expr { ... }`

  - can be used anywhere an expression can occur in Promela
- example

```
int x;
BEGIN_ACTIVE_MPI_PROC(proc, 2)
  int a[10];
  do
  :: c_expr { 3.14159*Pproc->a[i]*Pproc->a[i] < 1 } ->
     ...
  :: else ->
     x = c_expr { someFunction(Pproc->a, 10) };
     ...
  od
```

MPI functions

- basic functions
  - `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Pack`, `MPI_Unpack`
- blocking point-to-point functions
  - `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Sendrecv_replace`
- nonblocking functions
  - `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`, `MPI_Test`, `MPI_Request_free`, `MPI_Request_get_status`, `MPI_Waitany`, `MPI_Testany`, `MPI_Waitall`, `MPI_Testall`, `MPI_Waitsome`, `MPI_Testsome`, `MPI_Iprobe`, `MPI_Probe`, `MPI_Cancel`, `MPI_Test_cancelled`, `MPI_Send_init`, `MPI_Recv_init`, `MPI_Start`, `MPI_Startall`
- collective functions
  - `MPI_Barrier`, `MPI_Reduce`, `MPI_Allreduce`, `MPI_Bcast`

MPI functions: general syntax

- almost all functions take as their first argument the letter `P` followed by the process name
  - e.g., `Pproc`, `Pmaster`, `Pslave`, ...
  - this is for technical reasons dealing with interface to SPIN
- almost all parameters are C expressions
  - don't forget to pre-append `Pproc->` or `now.` to variables
  - exception: those of boolean type
    - e.g., flag in `MPI_Test`
- no communicator argument
  - for now, the only communicator is `MPI_COMM_WORLD`
  - multiple communicators...coming

MPI_Init and MPI_Finalize

- `MPI_Init(Proc, rank)`

  `Proc` `P` followed by proctype name, e.g. `Pproducer`

  `rank` rank to assign to this process (C expression)

  - rank is usually a function of the pid
  - examples
    - `MPI_Init(Pproc, Pproc->_pid)`
    - `MPI_Init(Pslave, Pslave->_pid-1)`
  - no two processes can have the same rank
  - user must ensure ranks are $\{0, 1, \ldots,$ `NPROCS` $- 1\}$
- `MPI_Finalize(Proc)`

MPI_Send

- `MPI_Send(Proc, buffer, count, datatype, dest, tag)`

`Proc`	`P` followed by proctype name
`buffer`	pointer to beginning of send buffer (C expression of type `void*`)
`count`	number of elements in send buffer (C expression of integer type)
`datatype`	an MPI datatype, e.g., `MPI_INT` (C expression of integer type)
`dest`	rank of the destination process (C expression of integer type)
`tag`	tag to associate to the message (C expression of integer type)

Some useful MPI constants

- `MPI_ANY_SOURCE`
- `MPI_ANY_TAG`
- `MPI_STATUS_IGNORE`
- `MPI_STATUSES_IGNORE`
- `MPI_REQUEST_NULL`
- `MPI_BYTE`
- `MPI_SHORT`
- `MPI_INT`
- `MPI_POINT`
- `MPI_SYMBOLIC`
- `MPI_PACKED`
- `MPI_SUM`
- `MPI_MAX`

inlines

```
inline norm(a, b, result) {
  result = a*a + b*b
}
```

- text inserted into calling point
- actual parameters subsituted for formal parameters
- no return value
- no local variables
- essentially a macro

Use of the C preprocessor

- cpp is run on the source files before SPIN parses them
- convenient for specifying values of parameters, etc.
- http://gcc.gnu.org/onlinedocs/cpp/
- `#define N 10`
- ```
  #define printState(i)                          \
    if                                           \
    :: i = 10 -> printf("state: %d", a[i])       \
    :: else -> printf("state: %d", b[i+2])       \
    fi
  ```
- ```
  #ifdef NCOMP
    ...
  #else
    ...
  #endif
  ```

Example: Diffusion

- `diffusion/diffusion_dl1.prom`