Basic Spin Manual

Gerard J. Holzmann

Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

Spin is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols. The system is described in a modeling language called PROMELA. The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendez±vous), or asynchronous (i.e., buffered).

Given a model system specified in PROMELA, *spin* can either perform random simulations of the system's execution or it can generate a C program that performs an efficient online verification of the system's correctness properties. During simulation and verification *spin* checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to verify the correctness of system invariants, it can find non± progress execution cycles, and it can verify correctness properties expressed in next±time free linear temporal logic formulae.

The verifier is setup to be efficient and to use a minimal amount of memory. An exhaustive verification performed by *spin* can establish with mathematical certainty whether or not a given behavior is error±free. Very large verification problems, that can ordinarily not be solved within the constraints of a given computer system, can be attacked with a frugal ''bit state storage'' technique, also known as *supertrace*. With this method the state space can be collapsed to a small number of bits per reachable system state, with minimal side±effects.

The first part of this memo gives an introduction to PROMELA, the second part discusses the usage of *spin*, and the third part contains a brief reference manual for PROMELA. In the appendix an example is used to illustrate the construction of a basic PROMELA model for *spin* verification.

This manual discusses only the basic use of *spin*. It does not discuss extensions to the language that are part of *spin* version 2.0 and beyond (see the notes at the end of this manual). It also does not discuss the builtin support for the verification of linear temporal logic formulae.

 A comparable tutorial on the use of *spin* can also be found in Computer Networks and ISDN Systems, 1993, Vol. 25, No. 9, pp. 981±1017.

Consult the `WhatsNew.ps` document for an overview of newer features of the *spin* system. See `Roadmap.ps` for a quick summary of the basic procedure of working with *spin*.

1. Introduction to PROMELA

PROMELA is a verification modeling language. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of *spin* is to verify fractions of process behavior, that for one reason or another are considered suspect. The relevant behavior is modeled in PROMELA and verified. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed PROMELA models. Each model can be verified with *spin* under different types of assumptions about the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with *spin*, that fact can be used in the construction and verification of all subsequent models.

PROMELA programs consist of *processes*, message *channels*, and *variables*. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

Executability

In PROMELA there is no difference between conditions and statements, even isolated boolean conditions can be used as statements. The execution of every statement is conditional on its *executability*. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. For instance, instead of writing a busy wait loop:

```
while (a != b)
    skip    /* wait for a==b */
```

one can achieve the same effect in PROMELA with the statement

```
(a == b)
```

A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does.

Variables are used to store either global information about the system as a whole, or information local to one specific process, depending on where the declaration for the variable is placed. The declarations

```
bool flag;
int state;
byte msg;
```

define variables that can store integer values in three different ranges. The scope of a variable is global if it is declared outside all process declarations, and local if it is declared within a process declaration.

Data Types

Table 1 summarizes the basic data types, sizes, and the corresponding value ranges (on a DEC VAX computer).

Table 1 ± Data Types

Name	Size (bits)	Usage	Range
bit	1	unsigned	0..1
bool	1	unsigned	0..1
byte	8	unsigned	0..255
mtype	8	unsigned	0..255
short	16	signed	±2^15..2^15±1
int	32	signed	±2^31..2^31±1

The names `bit` and `bool` are synonyms for a single bit of information. A `byte` is an unsigned quantity that can store a value between `0` and `255`. `short`s and `int`s are signed quantities that differ only in the range of values they can hold.

An `mtype` variable can be assigned symbolic values that are declared in an `mtype = { ... }` statement, to be discussed below.

Array Variables

Variables can be declared as arrays. For instance,

```
byte state[N]
```

declares an array of `N` bytes that can be accessed in statements such as

```
state[0] = state[3] + 5 * state[3*2/n]
```

where `n` is a constant or a variable declared elsewhere. The index to an array can be any expression that determines a unique integer value. The effect of an index value outside the range `"0`.. N±1" is undefined; most likely it will cause a runtime error. (Multi±dimensional arrays can be defined indirectly with the help of the `typedef` construct, see `WhatsNew.ps`, Section 2.1.7)

So far we have seen examples of variable declarations and of two types of statements: boolean conditions and assignments. Declarations and assignments are always *executable*. Conditions are only executable when they hold.

Process Types

The state of a variable or of a message channel can only be changed or inspected by processes. The behavior of a process is defined in a `proctype` declaration. The following, for instance, declares a process with one local variable `state`.

```
proctype A()
{   byte state;

    state = 3
}
```

The process type is named `A`. The body of the declaration is enclosed in curly braces. The declaration body consists of a list of zero or more declarations of local variables and/or statements. The declaration above contains one local variable declaration and a single statement: an assignment of the value 3 to variable `state`.

The semicolon is a statement *separator* (not a statement terminator, hence there is no semicolon after the last statement). PROMELA accepts two different statement separators: an arrow `'±>'`and the semicolon `';'`. The two statement separators are equivalent. The arrow is sometimes used as an informal way to indicate a causal relation between two statements. Consider the following example.

```
byte state = 2;

proctype A()
{   (state == 1) ±> state = 3
}
proctype B()
{   state = state ± 1
}
```

In this example we declared two types of processes, `A` and `B`. Variable `state` is now a global, initialized to the value two. Process type `A` contains two statements, separated by an arrow. In the example, process declaration `B` contains a single statement that decrements the value of the state variable by one. Since the assignment is always executable, processes of type `B` can always complete without delay. Processes of type `A`, however, are delayed at the condition until the variable `state` contains the proper value.

Process Instantiation

A `proctype` definition only declares process behavior, it does not execute it. Initially, in the PROMELA model, just one process will be executed: a process of type `init`, that must be declared explicitly in every PROMELA specification. The smallest possible PROMELA specification, therefore, is:

```
init { skip }
```

where `skip` is a dummy, null statement. More interestingly, however, the initial process can initialize global variables, and instantiate processes. An `init` declaration for the above system, for instance, could look as follows.

```
init
{   run A(); run B()
}
```

`run` is used as a unary operator that takes the name of a process type (e.g. `A`). It is executable only if a process of the type specified can be instantiated. It is unexecutable if this cannot be done, for instance if too many processes are already running.

The run statement can pass parameter values of all basic data types to the new process. The declarations are then written, for instance, as follows:

```
proctype A(byte state; short foo)
{
    (state == 1) ±> state = foo
}
init
{
    run A(1, 3)
}
```

Data arrays or process types can not be passed as parameters. As we will see below, there is just one other data type that can be used as a parameter: a message channel.

`Run` statements can be used in any process to spawn new processes, not just in the initial process. Processes are created with the `run` statements. An executing process disappears again when it terminates (i.e., reaches the end of the body of its process type declaration), but not before all processes that it started have terminated.

With the `run` statement we can create any number of copies of the process types `A` and `B`. If, however, more than one concurrent process is allowed to both read and write the value of a global variable a well± known set of problems can result; for example see [2]. Consider, for instance, the following system of two processes, sharing access to the global variable `state`.

```
byte state = 1;

proctype A()
{   (state==1) ±> state = state+1
}
proctype B()
{   (state==1) ±> state = state±1
}
init
{   run A(); run B()
}
```

If one of the two processes completes before its competitor has started, the other process will block forever on the initial condition. If both pass the condition simultaneously, both will complete, but the resulting value of `state` is unpredictable. It can be any of the values `0`, `1`, or `2`.

Many solutions to this problem have been considered, ranging from an abolishment of global variables to the provision of special machine instructions that can guarantee an indivisible test and set sequence on a shared variable. The example below was one of the first solutions published. It is due to the Dutch mathematician Dekker. It grants two processes mutually exclusion access to an arbitrary *critical section* in their code, by manipulation three additional global variables. The first four lines in the PROMELA specification below are C±style macro definitions. The first two macros define `true` to be a constant value equal to `1` and `false` to be a constant `0`. Similarly, `Aturn` and `Bturn` are defined as constants.

```
#define true    1
#define false   0
#define Aturn   false
#define Bturn   true

bool x, y, t;
```

```
proctype A()
{   x = true;
    t = Bturn;
    (y == false || t == Aturn);
    /* critical section */
    x = false
}
proctype B()
{   y = true;
    t = Aturn;
    (x == false || t == Bturn);
    /* critical section */
    y = false
}
init
{   run A(); run B()
}
```

The algorithm can be executed repeatedly and is independent of the relative speeds of the two processes.

Atomic Sequences

In PROMELA there is also another way to avoid the *test and set* problem: `atomic` sequences. By prefixing a sequence of statements enclosed in curly braces with the keyword `atomic` the user can indicate that the sequence is to be executed as one indivisible unit, non±interleaved with any other processes. It causes a run±time error if any statement, other than the first statement, blocks in an atomic sequence. This is how we can use atomic sequences to protect the concurrent access to the global variable `state` in the earlier example.

```
byte state = 1;

proctype A()
{   atomic {
      (state==1) ±> state = state+1
    }
}
proctype B()
{   atomic {
      (state==1) ±> state = state±1
    }
}
init
{   run A(); run B()
}
```

In this case the final value of `state` is either zero or two, depending on which process executes. The other process will be blocked forever.

Atomic sequences can be an important tool in reducing the complexity of verification models. Note that atomic sequence restricts the amount of interleaving that is allowed in a distributed system. Otherwise untractable models can be made tractable by, for instance, labeling all manipulations of local variables with atomic sequences. The reduction in complexity can be dramatic.

Message Passing

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance as follows:

```
chan qname = [16] of { short }
```

This declares a channel that can store up to 16 messages of type `short`. Channel names can be passed from one process to another via channels or as parameters in process instantiations. If the messages to be passed by the channel have more than one field, the declaration may look as follows:

```
chan qname = [16] of { byte, int, chan, byte }
```

This time the channel stores up to sixteen messages, each consisting of two 8±bit values, one 32±bit value,

and a channel name.

The statement

```
qname!expr
```

sends the value of expression `expr` to the channel that we just created, that is: it appends the value to the tail of the channel.

```
qname?msg
```

receives the message, it retrieves it from the head of the channel, and stores it in a variable `msg`. The channels pass messages infirst±in±first±outorder. In the above cases only a single value is passed through the channel. If more than one value is to be transferred per message, they are specified in a comma separated list

```
qname!expr1,expr2,expr3
qname?var1,var2,var3
```

If more parameters are sent per message then the message channel can store, the redundant parameters are lost without warning. If fewer parameters are sent then the message channel can store, the value of the remaining parameters is undefined. Similarly, of the receive operations tries to retrieve more parameters than available, the value of the extra parameters is undefined; if it receives fewer than the number of parameters that was sent, the extra information is lost.

By convention, the first message field is often used to specify the message type (i.e. a constant). An alternative, and equivalent, notation for the send and receive operations is therefore to specify the message type, followed by a list of message fields enclosed in braces. In general:

```
qname!expr1(expr2,expr3)
qname?var1(var2,var3)
```

The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non empty. Optionally, some of the arguments of the receive operation can be constants:

```
qname?cons1,var2,cons2
```

in this case, a further condition on the executability of the receive operation is that the value of all message fields that are specified as constants match the value of the corresponding fields in the message that is at the head of the channel. Again, nothing bad will happen if a statement happens to be non±executable. The process trying to execute it will be delayed until the statement, or, more likely, an alternative statement, becomes executable.

Here is an example that uses some of the mechanisms introduced so far.

```
proctype A(chan q1)
{   chan q2;
    q1?q2;
    q2!123
}
proctype B(chan qforb)
{   int x;
    qforb?x;
    printf("x = %d\n", x)
}
init {
    chan qname = [1] of { chan };
    chan qforb = [1] of { int };
    run A(qname);
    run B(qforb);
    qname!qforb
}
```

The value printed will be `123`.

A predefined function `len(qname)` returns the number of messages currently stored in channel `qname`. Note that if `len` is used as a statement, rather than on the right hand side of an assignment, it will be unexecutable if the channel is empty: it returns a zero result, which by definition means that the statement is temporarily unexecutable. Composite conditions such as

```
(qname?var == 0)
```
or
```
(a > b && qname!123)
```
are invalid in PROMELA (note that these conditions can not be evaluated without side±effects). For a receive statement there is an alternative, using square brackets around the clause behind the question mark.
```
qname?[ack,var]
```
is evaluated as a condition. It returns `1` if the corresponding receive statement
```
qname?ack,var
```
is executable, i.e., if there is indeed a message `ack` at the head of the channel. It returns `0` otherwise. In neither case has the evaluation of a statement such as
```
qname?[ack,var]
```
any side±effects: the receive is evaluated, not executed.

Note carefully that in non±atomic sequences of two statements such as
```
(len(qname) < MAX) ±> qname!msgtype
```
or
```
qname?[msgtype] ±> qname?msgtype
```
the second statement is not *necessarily* executable after the first one has been executed. There may be race conditions if access to the channels is shared between several processes. In the first case another process can send a message to channel `qname` just after this process determined that the channel was not full. In the second case, the other process can steal away the message just after our process determined its presence.

Rendez±VousCommunication

So far we have talked about asynchronous communication between processes via message channels, declared in statements such as
```
chan qname = [N] of { byte }
```
where `N` is a positive constant that defines the buffer size. A logical extension is to allow for the declaration
```
chan port = [0] of { byte }
```
to define a rendez±vous port that can pass single byte messages. The channel size is zero, that is, the channel `port` can pass, but can not store messages. Message interactions via such rendez±vous ports are by definition synchronous. Consider the following example.
```
#define msgtype 33

chan name = [0] of { byte, byte };

proctype A()
{   name!msgtype(124);
    name!msgtype(121)
}
proctype B()
{   byte state;
    name?msgtype(state)
}
init
{   atomic { run A(); run B() }
}
```
Channel `name` is a global rendez±vous port. The two processes will synchronously execute their first statement: a handshake on message `msgtype` and a transfer of the value 124 to local variable `state`. The second statement in process `A` will be unexecutable, because there is no matching receive operation in process `B`.

If the channel `name` is defined with a non±zero buffer capacity, the behavior is different. If the buffer size is at least 2, the process of type `A` can complete its execution, before its peer even starts. If the buffer size is 1, the sequence of events is as follows. The process of type `A` can complete its first send action, but it blocks on the second, because the channel is now filled to capacity. The process of type `B` can then retrieve the first message and complete. At this point `A` becomes executable again and completes, leaving its last message as a residual in the channel.

Rendez±vous communication is binary: only two processes, a sender and a receiver, can be synchronized in a rendez±vous handshake. We will see an example of a way to exploit this to build a semaphore below. But first, let us introduce a few more control flow structures that may be useful.

2. Control Flow

Between the lines, we have already introduced three ways of defining control flow: concatenation of statements within a process, parallel execution of processes, and atomic sequences. There are three other control flow constructs in PROMELA to be discussed. They are case selection, repetition, and unconditional jumps.

Case Selection

The simplest construct is the selection structure. Using the relative values of two variables `a` and `b` to choose between two options, for instance, we can write:

```
if
:: (a != b) ±> option1
:: (a == b) ±> option2
fi
```

The selection structure contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard*.

In the above example the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard. The following example, for instance, uses input statements.

```
#define a 1
#define b 2

chan ch = [1] of { byte };

proctype A()
{   ch!a
}
proctype B()
{   ch!b
}
proctype C()
{   if
    :: ch?a
    :: ch?b
    fi
}
init
{   atomic { run A(); run B(); run C() }
}
```

The example defines three processes and one channel. The first option in the selection structure of the process of type `C` is executable if the channel contains a message `a`, where `a` is a constant with value `1`, defined in a macro definition at the start of the program. The second option is executable if it contains a message `b`, where, similarly, `b` is a constant. Which message will be available depends on the unknown relative speeds of the processes.

A process of the following type will either increment or decrement the value of variable `count` once.

```
byte count;

proctype counter()
{
    if
    :: count = count + 1
    :: count = count ± 1
    fi
}
```

Repetition

A logical extension of the selection structure is the repetition structure. We can modify the above program as follows, to obtain a cyclic program that randomly changes the value of the variable up or down.

```
byte count;

proctype counter()
{
    do
    :: count = count + 1
    :: count = count ± 1
    :: (count == 0) ±> break
    od
}
```

Only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated. The normal way to terminate the repetition structure is with a `break` statement. In the example, the loop can be broken when the count reaches zero. Note, however, that it need not terminate since the other two options always remain executable. To force termination we could modify the program as follows.

```
proctype counter()
{
    do
    :: (count != 0) ±>
        if
        :: count = count + 1
        :: count = count ± 1
        fi
    :: (count == 0) ±> break
    od
}
```

Unconditional Jumps

Another way to break the loop is with an unconditional jump: the infamous `goto` statement. This is illustrated in the following implementation of Euclid's algorithm for finding the greatest common divisor of two non±zero, positive numbers:

```
proctype Euclid(int x, y)
{
    do
    :: (x >  y) ±> x = x ± y
    :: (x <  y) ±> y = y ± x
    :: (x == y) ±> goto done
    od;
done:
    skip
}
```

The `goto` in this example jumps to a label named `done`. A label can only appear before a statement. Above we want to jump to the end of the program. In this case a dummy statement `skip` is useful: it is a place holder that is always executable and has no effect. The `goto` is also always executable.

The following example specifies a filter that receives messages from a channel `in` and divides them over two channels `large` and `small` depending on the values attached. The constant `N` is defined to be `128` and `size` is defined to be `16` in the two macro definitions.

```
#define N    128
#define size  16

chan in    = [size] of { short };
chan large = [size] of { short };
chan small = [size] of { short };

proctype split()
{   short cargo;

    do
    :: in?cargo ±>
        if
        :: (cargo >= N) ±>
            large!cargo
        :: (cargo <  N) ±>
            small!cargo
        fi
    od
}
init
{   run split()
}
```

A process type that merges the two streams back into one, most likely in a different order, and writes it back into the channel `in` could be specified as follows.

```
proctype merge()
{   short cargo;

    do
    ::  if
        :: large?cargo
        :: small?cargo
        fi;
        in!cargo
    od
}
```

If we now modify the `init` process as follows, the split and merge processes could busily perform their duties forever on.

```
init
{   in!345; in!12; in!6777;
    in!32;  in!0;
    run split();
    run merge()
}
```

As a final example, consider the following implementation of a Dijkstra semaphore, using binary rendez± vous communication.

```
#define p   0
#define v   1

chan sema = [0] of { bit };
```

```
proctype dijkstra()
{   byte count = 1;

    do
    :: (count == 1) ±>
        sema!p; count = 0
    :: (count == 0) ±>
        sema?v; count = 1
    od
}
proctype user()
{   do
    :: sema?p;
       /*     critical section */
       sema!v;
       /* non±critical section */
    od
}
init
{   run dijkstra();
    run user();
    run user();
    run user()
}
```

The semaphore guarantees that only one of the user processes can enter its critical section at a time. It does not necessarily prevent the monopolization of the access to the critical section by one of the processes.

Modeling Procedures and Recursion

Procedures can be modeled as processes, even recursive ones. The return value can be passed back to the calling process via a global variable, or via a message. The following program illustrates this.

```
proctype fact(int n; chan p)
{   chan child = [1] of { int };
    int result;

    if
    :: (n <= 1) ±> p!1
    :: (n >= 2) ±>
        run fact(n±1, child);
        child?result;
        p!n*result
    fi
}
init
{   chan child = [1] of { int };
    int result;

    run fact(7, child);
    child?result;
    printf("result: %d\n", result)
}
```

The process *fact(n, p)* recursively calculates the factorial of *n*, communicating the result via a message to its parent process *p*.

Timeouts

We have already discussed two types of statement with a predefined meaning in PROMELA: `skip`, and `break`. Another predefined statement is `timeout`. The `timeout` models a special condition that allows a process to abort the waiting for a condition that may never become true, e.g. an input from an empty channel. The timeout keyword is a modeling feature in PROMELA that provides an escape from a

hang state. The timeout condition becomes true only when no other statements within the distributed system is executable. Note that we deliberately abstract from absolute timing considerations, which is crucial in verification work, and we do not specify how the timeout should be implemented. A simple example is the following process that will send a reset message to a channel named *guard* whenever the system comes to a standstill.

```
proctype watchdog()
{
    do
    :: timeout ±> guard!reset
    od
}
```

Assertions

Another important language construct in PROMELA that needs little explanation is the `assert` statement. Statements of the form

```
assert(any_boolean_condition)
```

are always executable. If the boolean condition specified holds, the statement has no effect. If, however, the condition does not necessarily hold, the statement will produce an error report during verifications with *spin*.

3. More Advanced Usage

The modeling language has a few features that specifically address the verification aspects. It shows up in the way labels are used, in the semantics of the PROMELA `timeout` statement, and in the usage of statements such as `assert` that we discuss next.

End±StateLabels

When PROMELA is used as a verification language the user must be able to make very specific assertions about the behavior that is being modeled. In particular, if a PROMELA is checked for the presence of deadlocks, the verifier must be able to distinguish a normal *end state* from an abnormal one.

A normal end state could be a state in which every PROMELA process that was instantiated has properly reached the end of the defining program body, and all message channels are empty. But, not all PROMELA process are, of course, meant to reach the end of their program body. Some may very well linger in an `IDLE` state, or they may sit patiently in a loop ready to spring into action when new input arrives.

To make it clear to the verifier that these alternate end states are legal, and do not constitute a deadlock, a PROMELA model can use end state labels. For instance, if by adding a label to the process type `dijkstra()`, from section 1.9:

```
proctype dijkstra()
{   byte count = 1;

end:    do
    :: (count == 1) ±>
        sema!p; count = 0
    :: (count == 0) ±>
        sema?v; count = 1
    od
}
```

we indicate that it is not an error if, at the end of an execution sequence, a process of type `dijkstra()` has not reached its closing curly brace, but waits in the loop. Of course, such a state could still be part of a deadlock state, but if so, it is not caused by this particular process. (It will still be reported if any one of the other processes in not in a valid end±state).

There may be more than one end state label per verification model. If so, all labels that occur within the same process body must be unique. The rule is that every label name that *starts* with the three character sequence `"end"` is an endstate label. So it is perfectly valid to use variations such as `enddne`, `end0`, `end_appel`, etc.

Progress±StateLabels

In the same spirit as the end state labels, the user can also define *progress state* labels. In this case, a progress state labels will mark a state that *must* be executed for the protocol to make progress. Any infinite cycle in the protocol execution that does not pass through at least one of these progress states, is a potential starvation loop. In the `dijkstra` example, for instance, we can label the successful passing of a semaphore test as ''progress'' and ask a verifier to make sure that there is no cycle in the protocol execution where at least one process succeeds in passing the semaphore guard.

```
proctype dijkstra()
{   byte count = 1;

end:    do
    :: (count == 1) ±>
progress:   sema!p; count = 0
    :: (count == 0) ±>
        sema?v; count = 1
    od
}
```

If more than one state carries a progress label, variations with a common prefix are again valid: `progress0`, `progress_foo`, etc.

All analyzers generated by *spin* with the `±a` flag have a runtime option (after compilation) named `±l`. Invoking the generated analyzer with that flag will cause a fast search for non±progress loops, instead of the default search for deadlocks. The search takes about twice as long (and uses twice as much memory) as the default search for deadlocks. (A considerable improvement over standard methods that are based on the analysis of strongly connected components.)

Message Type Definitions

We have seen how variables are declared and how constants can be defined using C±style macros. PROMELA also allows for message type definitions that look as follows:

```
mtype = {
    ack, nak, err,
    next, accept
}
```

This is a preferred way of specifying the message types since it abstracts from the specific values to be used, and it makes the names of the constants available to an implementation, which can improve error reporting.

By using the `mtype` keyword in channel declarations, the corresponding message field will always be interpreted symbolically, instead of numerically. For instance:

```
chan q = [4] of { mtype, mtype, bit, short };
```

Pseudo Statements

We have now discussed all the basic types of statements defined in PROMELA: assignments, conditions, send and receive, `assert`, `timeout`, `goto`, `break` and `skip`. Note that `chan`, `len` and `run` are not really statements but unary operators that can be used in conditions and assignments.

The `skip` statement was mentioned in passing as a statement that can be a useful filler to satisfy syntax requirements, but that really has no effect. It is formally not part of the language but a *pseudo±statement*, merely a synonym of another statement with the same effect: a simple condition of a constant value `(1)`. In the same spirit other pseudo±statements could be defined (but are not), such as `block` or `hang`, as equivalents of `(0)`, and `halt`, as an equivalent of `assert(0)`.. Another pseudo±statement is `else`, that can be used as the initial statement of the last option sequence in a selection or iteration.

```
if
:: a > b ±> ...
:: else ±> ...
fi
```

The `else` is only executable (true) if all other options in the same selection are not executable.

Example

Here is a simple example of a (flawed) protocol, modeled in PROMELA.

```
mtype = { ack, nak, err, next, accept };

proctype transfer(chan in,out,chin,chout)
{   byte o, i;

    in?next(o);
    do
    :: chin?nak(i) ±>
            out!accept(i);
            chout!ack(o)
    :: chin?ack(i) ±>
            out!accept(i);
            in?next(o);
            chout!ack(o)
    :: chin?err(i) ±>
            chout!nak(o)
    od
}

init
{   chan AtoB = [1] of { mtype, byte };
    chan BtoA = [1] of { mtype, byte };
    chan Ain  = [2] of { mtype, byte };
    chan Bin  = [2] of { mtype, byte };
    chan Aout = [2] of { mtype, byte };
    chan Bout = [2] of { mtype, byte };
    atomic {
      run transfer(Ain,Aout, AtoB,BtoA);
      run transfer(Bin,Bout, BtoA,AtoB)
    };
    AtoB!err(0)
}
```

The channels `Ain` and `Bin` are to be filled with token messages of type `next` and arbitrary values (e.g. ASCII character values) by unspecified background processes: the users of the transfer service. Similarly, these user processes can read received data from the channels `Aout` and `Bout`. The channels and processes are initialized in a single atomic statement, and started with the dummy `err` message.

4. Introduction to Spin

Given a model system specified in PROMELA, *spin* can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. The verifier can check, for instance, if user specified system invariants may be violated during a protocol's execution.

If *spin* is invoked without any options it performs a random simulation. With option `±n`*N* the seed for the simulation is set explicitly to the integer value *N*.

A group of options `pglrs` can be used to set the desired level of information that the user wants about the simulation run. Every line of output normally contains a reference to the source line in the specification that caused it.

`±p` Shows the state changes of the PROMELA processes at every time step.

`±g` Shows the current value of global variables at every time step.

`±l` Shows the current value of local variables, after the process that owns them has changed state. It

is best used in combination with option ±p.

±r Shows all message receive events. It shows the process performing the receive, its name and number, the source line number, the message parameter number (there is one line for each parameter), the message type and the message channel number and name.

±s Shows all message send events. *Spin* understands three other options (±a, ±m, and ±t):

±a Generates a protocol specific analyzer. The output is written into a set of C files, named pan.[cbhmt], that can be compiled to produce the analyzer (which is then executed to perform the analysis). To guarantee an exhaustive exploration of the state space, the program can be compiled simply as

```
$ cc ±o run pan.c
```

For larger systems this may, however, exhaust the available memory on the machine used. Large to very large systems can still be analyzed by using a memory efficient bit state space method by

```
$ cc ±DBITSTATE ±o run pan.c
```

An indication of the coverage of such a search can be derived from the *hash factor* (see below). The generated executable analyzer, named run above, has its own set of options that can be seen by typing "run±?" (see also below in *Using the Analyzer*).

±m can be used to change the default semantics of send actions. Normally, a send operation is only executable if the target channel is non±full. This imposes an implicit synchronization that can not always be justified. Option ±m causes send actions to be always executable. Messages sent to a channel that is full are then dropped. If this option is combined with ±a the semantics of send in the analyzers generated is similarly altered, and the verifications will take the effects of this type of message loss into consideration.

±t is a trail±hunting option. If the analyzer finds a violation of an assertion, a deadlock or an unspecified reception, it writes an error trail into a file named pan.trail. The trail can be inspected in detail by invoking *spin* with the ±t option. In combination with the options pglrs different views of the error sequence are then easily obtained.

For brevity, other options of *spin* are not discussed here. For details see [5]. For a hint of their purpose, see ‘‘Digging Deeper’’ at the end of this manual.

The Simulator

Consider the following example protocol, that we will store in a file named lynch.

```
 1  #define MIN   9
 2  #define MAX   12
 3  #define FILL  99
 4
 5  mtype = { ack, nak, err }
 6
 7  proctype transfer(chan chin, chout)
 8  { byte o, i, last_i=MIN;
 9
10    o = MIN+1;
11    do
12    :: chin?nak(i) ±>
13        assert(i == last_i+1);
14        chout!ack(o)
```

```
15    :: chin?ack(i) ±>
16        if
17        :: (o <  MAX) ±> o = o+1
18        :: (o >= MAX) ±> o = FILL
19        fi;
20        chout!ack(o)
21    :: chin?err(i) ±>
22        chout!nak(o)
23    od
24  }
25
26  proctype channel(chan in, out)
27  { byte md, mt;
28    do
29    :: in?mt,md ±>
30        if
31        :: out!mt,md
32        :: out!err,0
33        fi
34    od
35  }
36
37  init
38  { chan AtoB = [1] of { mtype, byte };
39    chan BtoC = [1] of { mtype, byte };
40    chan CtoA = [1] of { mtype, byte };
41    atomic {
42        run transfer(AtoB, BtoC);
43        run channel(BtoC, CtoA);
44        run transfer(CtoA, AtoB)
45    };
46    AtoB!err,0; /* start */
47    0       /* hang */
48  }
```

The protocol uses three message types: *ack*, *nak*, and a special type *err* that is used to model message distortions on the communication channel between the two transfer processes. The behavior of the channel is modeled explicitly with a channel process. There is also an assert statement that claims a, faulty, invariant relation between two local variables in the transfer processes.

Running *spin* without options gives us a random simulation that will only provide output when execution terminates, or if a *printf* statement is encountered. In this case:

```
$ spin lynch
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer)  line 11 (state 15)
proc  2 (channel)   line 28 (state 6)
proc  1 (transfer)  line 13 (state 3)
proc  0 (:init:)    line 48 (state 6)
4 processes created
$
```

There are no *printf*'s in the specification, but execution halts on an assertion violation. Curious to find out more, we can repeat the run with more verbose output, e.g. printing all receive events. The result of that run is shown in Figure 1. Most output will be self±explanatory.

The above simulation run ends in the same assertion violation. Since the simulation resolves nondeterministic choices in a random manner, this need not always be the case. To force a reproducible run, the option ±n*N* can be used. For instance:

```
$ spin ±r ±n100 lynch
```

will seed the random number generator with the integer value 100 and is guaranteed to produce the same

output each time it is executed.

The other options can add still more output to the simulation run, but the amount of text can quickly become overwhelming. An easy solution is to filter the output through *grep*. For instance, if we are only interested in the behavior of the channel process in the above example, we say:

```
$ spin ±n100 ±r lynch | grep "proc  2"
```

The results are shown in Figure 1.

```
$ spin ±r lynch
proc  1 (transfer) line  21, Recv err,0  <± queue 1 (chin)
proc  2 (channel)  line  29, Recv nak,10 <± queue 2 (in)
proc  3 (transfer) line  12, Recv nak,10 <± queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,10 <± queue 1 (chin)
...
proc  1 (transfer) line  15, Recv ack,12 <± queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <± queue 2 (in)
proc  3 (transfer) line  15, Recv ack,99 <± queue 3 (chin)
proc  1 (transfer) line  15, Recv ack,99 <± queue 1 (chin)
proc  2 (channel)  line  29, Recv ack,99 <± queue 2 (in)
proc  3 (transfer) line  21, Recv err,0  <± queue 3 (chin)
proc  1 (transfer) line  12, Recv nak,99 <± queue 1 (chin)
spin: "lynch" line 13: assertion violated
#processes: 4
proc  3 (transfer) line 11 (state 15)
proc  2 (channel)  line 28 (state 6)
proc  1 (transfer) line 13 (state 3)
proc  0 (:init:)   line 48 (state 6)
4 processes created
$ spin ±n100 ±r lynch | grep "proc  2"
proc  2 (channel) line 29, Recv nak,10 <± queue 2 (in)
proc  2 (channel) line 29, Recv ack,11 <± queue 2 (in)
proc  2 (channel) line 29, Recv ack,12 <± queue 2 (in)
proc  2 (channel) line 28 (state 6)
```

Figure 1. Simulation Run Output

The Analyzer

The simulation runs can be useful in quick debugging of new designs, but by simulation alone we can not prove that the system is really error free. A verification of even very large models can be performed with the `±a` and `±t` options of *spin*. (See also Roadmap.ps for quick guidelines on what to do.)

An exhaustive state space searching program for a protocol model is generated as follows, producing five files, named *pan.[bchmt]*.

```
$ spin ±a lynch
$ wc pan.[bchmt]
     99     285    1893 pan.b
   3158   10208   70337 pan.c
    356    1238    7786 pan.h
    216     903    6045 pan.m
    575    2099   14017 pan.t
   4404   14733  100078 total
```

The details are none too interesting: *pan.c* contains most of the C code for the analysis of the protocol. File *pan.t* contains a transition matrix that encodes the protocol control flow; *pan.b* and *pan.m* contain C code for forward and backward transitions and *pan.h* is a general header file. The program can be compiled in several ways, e.g., with a full state space or with a bit state space.

Exhaustive Search

The best method, that works up to system state spaces of roughly 100,000 states, is to use the default compilation of the program:

```
$ cc ±o run pan.c
```

The executable program *run* can now be executed to perform the verification. The verification is truly exhaustive: it tests all possible event sequences in all possible orders. It should, of course, find the same assertion violation.

```
$ run
assertion violated (i == last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/5)
```

(The output format of the more recent versions of *spin* is more elaborate, but it includes the same information.)

```
$ run
assertion violated (i == last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/5)
```

The first line of the output announces the assertion violation and attempts to give a first indication of the invariant that was violated. The violation was found after 61 states had been generated. Hash "conflicts" gives the number of hash collisions that happened during access to the state space. As indicated, all collissions are resolved in full search mode, since all states are placed in a linked list. The most relevant piece of output in this case, however, is on the third line which tells us that a trail file was created that can be used in combination with the simulator to recreate the error sequence. We can now say, for instance

```
$ spin ±t ±r lynch | grep "proc  2"
```

to determine the cause of the error. Note carefully that the verifier is guaranteed to find the assertion violation if it is feasible. If an exhaustive search does not report such a violation, it is certain that *no* execution execution sequence exists that can violate the assertion.

Options

The executable analyzer that is generated comes with a modest number of options that can be checked as follows

```
$ run ±±
±cN stop at Nth error (default=1)
±l  find non±progress cycles    # when compiled with ±DNP
±a  find acceptance cycles  # when not compiled with ±DNP
±mN max depth N (default=10k)
±wN hash table of 2^N entries (default=18)
 ...etc.
```

Using a zero as an argument to the first option forces the state space search to continue, even if errors are found. An overview of unexecutable (unreachable) code is given with every complete run: either the default run if it did not find any errors, or the run with option `±c0`. In this case the output is:

```
$ run ±c0
assertion violated (i == (last_i + 1))
```

```
vector 64 byte, depth reached 60, errors: 5
     165 states, stored
       5 states, linked
      26 states, matched
hash conflicts: 1 (resolved)
(size 2^18 states, stack frames: 0/6)

unreached code :init: (proc 0):
    reached all 9 states
unreached code channel (proc 1):
     line 35 (state 9),
    reached: 8 of 9 states
unreached code transfer (proc 2):
     line 24 (state 18),
    reached: 17 of 18 states
```

There were five assertion violations, and some 165 unique system states were generated. Each state description (the *vector size*) took up 64 bytes of memory; the longest non±cyclic execution sequence was 60. There is one unreachable state both in the channel process and in the transfer process. In both cases the unreachable state is the control flow point just after the do±loop in each process. Note that both loops are indeed meant to be non±terminating.

The `±l` option will cause the analyzer to search for non±progress loops rather than deadlocks or assertion violations. The option is explained in the section on ‘‘More Advanced Usage.’’

The executable analyzer has two other options. By default the search depth is restricted to a rather arbitrary 10,000 steps. If the depth limit is reached, the search is truncated, making the verification less than exhaustive. To make certain that the search is exhaustive, make sure that the "depth reached" notice is within the maximum search depth, and if not, repeat the analysis with an explicit `±m` argument.

The `±m` option can of course also be used to truncate the search explicitly, in an effort to find the shortest possible execution sequence that violates a given assertion. Such a truncated search, however, is not guaranteed to find every possible violation, even within the search depth.

The last option `±w`*N* can only affect the run time, not the scope, of an analysis with a full state space. This "hash table width" should normally be set equal to, or preferably higher than, the logarithm of the expected number of unique system states generated by the analyzer. (If it is set too low, the number of hash collisions will increase and slow down the search.) The default *N* of 18 handles up to 262,144 system states, which should suffice for almost all applications of a full state space analysis.

Bit State Space Analyses

It can easily be calculated what the memory requirements of an analysis with a full state space are [4]. If, as in the example we have used, the protocol requires 64 bytes of memory to encode one system state, and we have a total of 2MB of memory available for the search, we can store up to 32,768 states. The analysis fails if there are more reachable states in the system state space. So far, *spin* is the *only* verification system that can avoid this trap. All other existing automated verification system (irrespective on which formalism they are based) simply run out of memory and abort their analysis without returning a useful answer to the user.

The coverage of a conventional analysis goes down rapidly when the memory limit is hit, i.e. if there are twice as many states in the full state space than we can store, the effective coverage of the search is only 50% and so on. *Spin* does substantially better in those cases by using the bit state space storage method [4]. The bit state space can be included by compiling the analyzer as follows:

```
$ cc ±DBITSTATE ±o run pan.c
```

The analyzer compiled in this way should of course find the same assertion violation again:

```
$ run
assertion violated (i == ((last_i + 1))
pan: aborted
pan: wrote pan.trail
search interrupted
vector 64 byte, depth reached 56
      61 states, stored
       5 states, linked
       1 states, matched
hash factor: 67650.064516
(size 2^22 states, stack frames: 0/5)
$
```

In fact, for small to medium size problems there is very little difference between the full state space method and the bit state space method (with the exception that the latter is somewhat faster and uses substantially less memory). The big difference comes for larger problems. The last two lines in the output are useful in estimating the *coverage* of a large run. The maximum number of states that the bit state space can accommodate is written on the last line (here 2^{22} bytes or about 32 million bits = states). The line above it gives the *hash factor*: roughly equal to the maximum number of states divided by the actual number of states. A large hash factor (larger than 100) means, with high reliability, a coverage of 99% or 100%. As the hash factor approaches 1 the coverage approaches 0%.

Note carefully that the analyzer realizes a partial coverage *only* in cases where traditional verifiers are either unable to perform a search, or realize a far smaller coverage. In *no* case will *spin* produce an answer that is less reliable than that produced by other automated verification systems (quite on the contrary).

The object of a bit state verification is to achieve a hash factor larger than 100 by allocating the maximum amount of memory for the bit state space. For the best result obtainable: use the `±w`*N* option to size the state space to precisely the amount of real (not virtual) memory available on your machine. By default, *N* is 22, corresponding to a state space of 4MB. For example, if your machine has 128MB of real memory, you can use `±w27` to analyze systems with up to a billion reachable states.

5. PROMELA Reference Manual

This section describes the language PROMELA (version 1). As much as possible, the presentation follows the example from the C reference manuals [6]. It does not cover possible restrictions or extensions of specific implementations. The current implementation of *spin*, for instance, has an extra keyword `printf`, to access the corresponding library function.

Lexical Conventions

There are five classes of tokens: identifiers, keywords, constants, operators and statement separators. Blanks, tabs, newlines, and comments serve only to separate tokens. If more than one interpretation is possible, a token is taken to be the longest string of characters that can constitute a token.

Comments

Any string started with `/*` and terminated with `*/` is a comment. Comments may not be nested.

Identifiers

An identifier is a single letter, period, or underscore followed by zero or more letters, digits, periods, or underscores.

Keywords

The following identifiers are reserved for use as keywords. (Those with a star * attached are new in *spin* Version 2.0, and not discussed in this paper.)

```
active*     assert     atomic
bit         bool       break
byte        chan       d_step*
do          else*      empty*
enabled*    fi         full*
goto        hidden*    if
init        int        len
mtype       nempty*    never
nfull*      od         of
pc_value*   printf     proctype
run         short      skip
timeout     typedef*   unless*
xr*         xs*        _*
```

Constants

A constant is a sequence of digits representing a decimal integer. There are no floating point numbers in PROMELA. Symbolic names for constants can be defined in two ways. The first method is to use a C±style macro definition

```
#define NAME    value
```

The second method is to use the keyword `mtype` (see ‘‘declarations’’ below).

Expressions

The following operators can be used to build expressions.

```
+ ± * / %
> >= < <= == != !
&& ||
& | ~ >> <<
++ ±±
```

Most operators are binary. The logical negation *!* and the minus ± operator can be both unary and binary, depending on context. The ++ and ±± operators are unary suffix operators, as they are defined also in C. Expressions can be used, for instance, in assignments of the type "a= expression" , with a a variable.

There are also 5 unary operators that apply only to message channels:

```
len, empty*, nempty*, nfull*, full*
```

`len` measures the number of messages an existing channel holds. There is one unary operator that is used for process instantiations:

```
run
```

And, finally, there are two binary operators

```
! ?
```

which are used for sending and receiving messages (see below).

Declarations

Processes, channels, and variables must be declared before they can be used. Variables and channels can be declared either locally, within a process, or globally. A process can only be declared globally in a `proctype` declaration. Local declarations may appear anywhere in a process body.

Variables

A variable declaration is started by a keyword indicating the basic data type of the variable, `bit`, `bool`, `byte`, `short`, or `int`, followed by one or more identifiers, optionally followed by an initializer.

```
byte name1, name2 = 4, name3
```

By default all variables are initialized to zero. An initializer, if specified, must be a constant. The table below summarizes the width and attributes of the basic data types.

Name	Size (bits)	Usage
bit	1	unsigned
bool	1	unsigned
byte	8	unsigned
mtype	8	unsigned
short	16	signed
int	32	signed

The names *bit* and *bool* are synonyms for a single bit of information. A *byte* is an unsigned quantity that can store a value between 0 and 255. *Short*s and *int*s are signed quantities that differ only in the range of values they can hold.

An array of variables is declared as follows:

```
int name1[N]
```

where `N` is a constant. An array can have a just a single constant as an initializer. If specified it is used to initialize all elements of the array.

Symbolic names for constants, e.g. message types, can, optionally, be defined in a declaration of the type

```
mtype = { namelist }
```

where `namelist` is a comma separated list of symbolic names.

Message Channels

A message channel can be declared, for instance, as follows:

```
chan name = [N] of { short, short }
```

where `N` is a constant that specifies the maximum number of messages that can be stored in the channel. A list of one or more data types (or the channel type `chan`) enclosed in curly braces defines the type of the messages that can be passed through the channel. All channels are initialized to be empty.

Processes

A process declaration starts with the keyword `proctype` followed by a name, a list of formal parameters enclosed in round braces, and a sequence of statements and local variable declarations. The body of process declaration is enclosed in curly braces.

```
proctype name( /* parameter decls */ )
{
    /* statements */
}
```

Statements

There are twelve (fifteen*) types of statements:

```
assertion   assignment     atomic
break       declaration    expression
goto        receive        selection
repetition  send           timeout
unless*     sorted_send*   random_receive*
```

Each statement may be preceded by a label: a name followed by a colon. A statement can only be passed if it is executable. To determine its executability the statement can be evaluated: if evaluation returns a zero value the statement is blocked. In all other cases the statement is executable and can be passed. The act of passing the statement after a successful evaluation is called the ‘‘execution’’ of the statement. There is one so±called *pseudo*±statement `skip`, which is really a syntactic equivalent of `(1)`. `skip`, therefore, is a null statement; it is always executable. It has no effect when executed, but may be needed to satisfy syntax requirements. The evaluation of an assertion statement `assert(condition)` has no effect if the condition holds, but aborts the running process if evaluation of the condition returns a zero result (the boolean value ‘‘false’’).

Goto statements can be used to transfer control to any labeled statement within the same process or procedure. They also are always executable. Assignments have been discussed above, they are always executable. A declaration is also always executable. Expressions are only executable if they return a non± zero value. That is, the expression 0 (zero) is never executable, and similarly 1 always is executable. Below we consider the remaining statements: selection, repetition, send, receive, break, timeout, and atomic statements.

Selection

A selection statement is started with the keyword `if`, followed by a list of one or more ‘options’ and terminated with the keyword `fi`. Every ‘option’ is started with the flag `::` followed by any sequence of statements. One and only one option from a selection statement will be selected for execution. The first statement of an option determines whether the option can be selected or not. If more than one option is executable, one will be selected at random. Note that this randomness makes the language a nondeterministic one.

Repetition and Break

A repetition or `do` statement is similar to a selection statement, but is executed repeatedly until either a `break` statement is executed or a `goto` jump will transfer control outside the cycle. The keywords of the repetition statement are `do` and `od` instead of the `if` and `fi` of selection. The `break` statement will terminate the innermost repetition statement in which it is executed. The use of a break statement outside a repetition statement is illegal.

Atomic Sequences

The keyword `atomic` introduces an atomic sequence of statements, that is to be executed as one indivisible step. The syntax is as follows

```
atomic { sequence }
```

Logically the sequence of statements is now equivalent to one single statement. It is a run±time error if any statement that is part of an atomic sequence is found to be unexecutable. The safest is therefore to include only assignments and local conditions in atomic sequences, but no sends or receives. Labeling local computations as atomic can bring an important reduction of the complexity of a verification model.

Send

The syntax of a send statement is:

```
expr1!expr2
```

where `expr1` returns the identity of a channel, e.g. obtained from a `chan` operation, and `expr2` returns a value to be appended to the channel. The send statement is not executable (blocks) if the addressed

channel is full or does not exist. If more than one value is to be passed from sender to receiver, the expressions are written in a comma separated list:

```
expr1!expr2,expr3,expr4
```

Equivalently, this may be written

```
expr1!expr2(expr3,expr4) .
```

Receive

The syntax of the receive statement is:

```
expr1?name
```

where `expr1` returns the name of a channel and `name` is a variable or a constant. If a constant is specified the receive statement is only executable if the channel exists and the oldest message stored in the channel contains the same value. If a variable is specified, the receive statement is executable if the channel exists and contains any message at all. The variable in that case will receive the value of the message that is retrieved. If more than one value is sent per message, the receive statement also take a comma separated list of variables and constants

```
expr1?name1,name2,...
```

which again is syntactically equivalent to

```
expr1?name1(name2,...)
```

Each constant in this list puts an extra condition on the executability of the receive: it must be matched by the value of the corresponding message field of the message to be retrieved. The variable fields retrieve the values of the corresponding message fields on a receive.

Placing square brackets around the clause after the '?' in the receiver operation converts it into a condition, that is true only if the corresponding receive operation is executable. It can be used freely in any type of composite boolean condition, and it has no side±effects when evaluated.

A last type of operation allowed on channels is

```
len(expr)
```

where `expr` returns the identity of an instantiated channel. The operation returns the number of messages in the channel specified, or zero if the channel does not exist.

Timeout

The timeout condition is a modeling feature that by definition becomes true only if no statement in any of the running processes is executable. It has no effect when executed.

Macros and Include Files

The source text of a specification is processed by the C [6] preprocessor for macro±expansion and file inclusions.

6. Summary

In the first part of this manual we have introduced a notation for modeling concurrent systems, including but not limited to asynchronous data communication protocols, in a language named PROMELA. The language has several unusual features. All communication between processes takes place via either messages or shared variables. Both synchronous and asynchronous communication are modeled as two special cases of a general message passing mechanism. Every statement in PROMELA can potentially model delay: it is either executable or not, in most cases depending on the state of the environment of the running process. Process interaction and process coordination is thus at the very basis of the language. More about the design of PROMELA, of the verifier *spin*, and its application to protocol design, can be found in [5].

PROMELA is deliberately a verification modeling language, not a programming language. There are, for instance, no elaborate abstract data types, or more than a few basic types of variable. A verification model is an abstraction of a protocol implementation. The abstraction maintains the essentials of the process interactions, so that it can be studied in isolation. It suppresses implementation and programming detail.

The syntax of PROMELA expressions, declarations, and assignments is loosely based on the language `C`[6]. The language was influenced significantly by the ''guarded command languages'' of E.W. Dijkstra [1] and C.A.R. Hoare [3]. There are, however, important differences. Dijkstra's language had no primitives for

process interaction. Hoare's language was based exclusively on synchronous communication. Also in Hoare's language, the type of statements that could appear in the guards of an option was restricted. The semantics of the selection and cycling statements in PROMELA is also rather different from other guarded command languages: the statements are not aborted when all guards are false but they block: thus providing the required synchronization.

With minimal effort *spin* allows the user to generate sophisticated analyzers from PROMELA verification models. Both the *spin* software itself, and the analyzers it can generate, are written in ANSI C and are portable across systems. They can be scaled to fully exploit the physical limitations of the host computer, and deliver within those limits the best possible analyses that can be realized with the current state of the art in protocol analysis.

7. References

[1] Dijkstra, E.W. ''Guarded commands, non±determinacy and formal derivation of programs.'' CACM 18, 8 (1975), 453±457.
[2] Dijkstra, E.W. ''Solution to a problem in concurrent programming control.'' CACM 8, 9 (1965), 569.
[3] Hoare, C.A.R. ''Communicating Sequential Processes.'' CACM 21, 8 (1978), 666±677.
[4] Holzmann, G.J. ''Algorithms for automated protocol verification.'' AT&T Technical Journal 69, 1 (Jan/Feb 1990). Special Issue on Protocol Testing, Specification, Verification.
[5] Holzmann, G.J. ''Design and Verification of Protocols.'' (1990), Prentice Hall, Englewood Cliffs, NJ 07632.
[6] Kernighan, B.W. and Ritchie, D.M. ''The C Programming Language.'' 2nd ed. (1988), Prentice Hall, Englewood Cliffs, NJ 07632.

Appendix: Building A Verification Suite

The first order of business in using *spin* for a verification is the construction of a faithful model in PROMELA of the problem at hand. The language is deliberately kept small. The purpose of the modeling is to extract those aspects of the system that are relevant to the coordination problem being studied. All other details are suppressed. Formally: the model is a reduction of the system that needs to be equivalent to the full system only with respect to the properties that are being verified. Once a model has been constructed, it becomes the basis for the construction of a series of, what we may call, ‘‘verification suites’’ that are used to verify its properties. To build a verification suite we can prime the model with assertions. The assertions can formalize invariant relations about the values of variables or about allowable sequences of events in the model.

As a first example we take the following solution to the mutual exclusion problem, discussed earlier, published in 1966 by H. Hyman in the Communications of the ACM. It was listed, in pseudo Algol, as follows.

```
 1  Boolean array b(0;1) integer k, i,
 2  comment process i, with i either 0 or 1, and k = 1±i;
 3  C0:   b(i) := false;
 4  C1:   if k != i then begin;
 5  C2:   if notb(1±i) then go to C2;
 6        else k := i; go to C1 end;
 7        else critical section;
 8        b(i) := true;
 9        remainder of program;
10        go to C0;
11        end
```

The solution, as Dekker’s earlier solution, is for two processes, numbered 0 and 1. Suppose we wanted to prove that Hyman’s solution truly guaranteed mutually exclusive access to the critical section. Our first task is to build a model of the solution in PROMELA. While we’re at it, we can pick some more useful names for the variables that are used.

```
 1  bool want[2]; /* Bool array b */
 2  bool turn;    /* integer    k */
 3
 4  proctype P(bool i)
 5  {
 6    want[i] = 1;
 7    do
 8    :: (turn != i) ±>
 9        (!want[1±i]);
10        turn = i
11    :: (turn == i) ±>
12        break
13    od;
14    skip; /* critical section */
15    want[i] = 0
16  }
17
18  init { run P(0); run P(1) }
```

We can generate, compile, and run a verifier for this model, to see if there are any major problems, such as a global system deadlock.

```
$ spin ±a hyman0
$ cc ±o pan pan.c
$ pan
full statespace search for:
assertion violations and invalid endstates
vector 20 byte, depth reached 19, errors: 0
      79 states, stored
       0 states, linked
      38 states, matched    total: 117
hash conflicts: 4 (resolved)
(size 2^18 states, stack frames: 3/0)

unreached code _init (proc 0):
    reached all 3 states
unreached code P (proc 1):
    reached all 12 states
```

The model passes this first test. What we are really interested in, however, is if the algorithm guarantees mutual exclusion. There are several ways to proceed. The simplest is to just add enough information to the model that we can express the correctness requirement in a PROMELA assertion.

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ±>
10        (!want[1±i]);
11        turn = i
12    :: (turn == i) ±>
13        break
14    od;
15    skip; /* critical section */
16    cnt = cnt+1;
17    assert(cnt == 1);
18    cnt = cnt±1;
19    want[i] = 0
20  }
21
22  init { run P(0); run P(1) }
```

We have added a global variable `cnt` that is incremented upon each access to the critical section, and decremented upon each exit from it. The maximum value that this variable should ever have is 1, and it can only have this value when a process is inside the critical section.

```
$ spin ±a hyman1
$ cc ±o pan pan.c
$ pan
assertion violated (cnt==1)
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 20 byte, depth reached 25, errors: 1
     123 states, stored
       0 states, linked
      55 states, matched    total: 178
hash conflicts: 42 (resolved)
(size 2^18 states, stack frames: 3/0)
```

The verifier claims that the assertion can be violated. We can use the error trail to check it with *spin*'s `±t` option:

```
$ spin ±t ±p hyman1
proc  0 (_init) line 24 (state 2)
proc  0 (_init) line 24 (state 3)
proc  2 (P) line 8 (state 7)
proc  2 (P) line 9 (state 2)
proc  2 (P) line 10 (state 3)
proc  2 (P) line 11 (state 4)
proc  1 (P) line 8 (state 7)
proc  1 (P) line 12 (state 5)
proc  1 (P) line 15 (state 10)
proc  2 (P) line 8 (state 7)
proc  2 (P) line 12 (state 5)
proc  2 (P) line 15 (state 10)
proc  2 (P) line 16 (state 11)
proc  2 (P) line 17 (state 12)
proc  2 (P) line 18 (state 13)
proc  1 (P) line 16 (state 11)
proc  1 (P) line 17 (state 12)
spin: "hyman1" line 17: assertion violated
step 17, #processes: 3
        want[0] = 1
        _p[0] = 12
        turn[0] = 1
        cnt[0] = 2
proc  2 (P) line 18 (state 13)
proc  1 (P) line 17 (state 12)
proc  0 (_init) line 24 (state 3)
3 processes created
```

Here is another way to catch the error. We again lace the model with the information that will allow us to count the number of processes in the critical section.

```
 1  bool want[2];
 2  bool turn;
 3  byte cnt;
 4
 5  proctype P(bool i)
 6  {
 7    want[i] = 1;
 8    do
 9    :: (turn != i) ±>
10        (!want[1±i]);
11        turn = i
```

```
12    :: (turn == i) ±>
13        break
14    od;
15    cnt = cnt+1;
16    skip;   /* critical section */
17    cnt = cnt±1;
18    want[i] = 0
19  }
20
21  proctype monitor()
22  {
23    assert(cnt == 0 || cnt == 1)
24  }
25
26  init {
27    run P(0); run P(1); run monitor()
28  }
```

The invariant condition on the value of counter `cnt` is now place in a separate process `monitor()` (the name is immaterial). The extra process runs along with the two others. It will always terminate in one step, but it could execute that step at *any* time. The systems modeled by PROMELA and verified by *spin* are completely asynchronous. That means that the verification of *spin* take into account *all* possible relative timings of the three processes. In a full verification, the assertion therefore can be evaluated at any time during the lifetime of the other two processes. If the verifier reports that it is not violated we can indeed conclude that there is no execution sequence at all (no way to select relative speeds for the three processes) in which the assertion can be violated. The setup with the monitor process is therefore an elegant way to check the validity of a system invariant. The verification produces:

```
$ spin ±a hyman2
$ cc ±o pan pan.c
$ pan
assertion violated ((cnt==0)||(cnt==1))
pan: aborted (at depth 15)
pan: wrote pan.trail
full statespace search for:
assertion violations and invalid endstates
search was not completed
vector 24 byte, depth reached 26, errors: 1
     368 states, stored
       0 states, linked
     379 states, matched    total: 747
hash conflicts: 180 (resolved)
(size 2^18 states, stack frames: 4/0)
```

Because of the extra interleaving of the two processes with a third monitor, the number of system states that had to be searched has increased, but the error is again correctly reported.

Another Example

Not always can a correctness requirement be cast in terms of a global system invariant. Here is an example that illustrates this. It is a simple alternating bit protocol, modeling the possibility of message loss, and distortion, and extended with negative acknowledgements.

```
1  #define MAX   5
2
3  mtype = { mesg, ack, nak, err };
4
```

```
 5  proctype sender(chan in, out)
 6  { byte o, s, r;
 7
 8    o=MAX±1;
 9    do
10    :: o = (o+1)%MAX; /* next msg */
11  again: if
12         :: out!mesg(o,s) /* send */
13         :: out!err(0,0)  /* distort */
14         :: skip       /* or lose */
15         fi;
16         if
17         :: timeout   ±> goto again
18         :: in?err(0,0) ±> goto again
19         :: in?nak(r,0) ±> goto again
20         :: in?ack(r,0) ±>
21        if
22        :: (r == s) ±> goto progress
23        :: (r != s) ±> goto again
24        fi
25         fi;
26  progress: s = 1±s /* toggle seqno */
27    od
28  }
29
30  proctype receiver(chan in, out)
31  { byte i;     /* actual input   */
32    byte s;     /* actual seqno   */
33    byte es;    /* expected seqno */
34    byte ei;    /* expected input */
35
36    do
37    :: in?mesg(i, s) ±>
38        if
39        :: (s == es) ±>
40            assert(i == ei);
41  progress:     es = 1 ± es;
42            ei = (ei + 1)%MAX;
43            if
44    /* send,   */   :: out!ack(s,0)
45    /* distort */   :: out!err(0,0)
46    /* or lose */   :: skip
47            fi
48        :: (s != es) ±>
49            if
50    /* send,   */   :: out!nak(s,0)
51    /* distort */   :: out!err(0,0)
52    /* or lose */   :: skip
53            fi
54        fi
55    :: in?err ±>
56        out!nak(s,0)
57    od
58  }
59
60  init {
61    chan s_r = [1] of { mtype,byte,byte };
62    chan r_s = [1] of { mtype,byte,byte };
63    atomic {
```

```
64        run sender(r_s, s_r);
65        run receiver(s_r, r_s)
66    }
67  }
```

To test the proposition that this protocol will correctly transfer data, the model has already been primed for the first verification runs. First, the sender is setup to transfer an infinite series of integers as messages, where the value of the integers are incremented modulo `MAX`. The value of `MAX` is not really too interesting, as long as it is larger than the range of the sequence numbers in the protocol: in this case 2. We want to verify that data that is sent can only be delivered to the receiver without any deletions or reorderings, despite the possibility of arbitrary message loss. The assertion on line 40 verifies precisely that. Note that if it were ever possible for the protocol to fail to meet the above requirement, the assertion can be violated.

A first verification run reassures us that this is not possible.

```
$ spin ±a ABP0
$ cc ±o pan pan.c
$ pan
full statespace search for:
assertion violations and invalid endstates
vector 40 byte, depth reached 131, errors: 0
     346 states, stored
       1 states, linked
     125 states, matched     total: 472
hash conflicts: 17 (resolved)
(size 2^18 states, stack frames: 0/25)

unreached code _init (proc 0):
    reached all 4 states
unreached code receiver (proc 1):
    line 58 (state 24)
    reached: 23 of 24 states
unreached code sender (proc 2):
    line 28 (state 27)
    reached: 26 of 27 states
```

But, be careful. The result means that all data that is delivered, is delivered in the correct order without deletions etc. We did not check that the data *will* necessarily be delivered. It may be possible for sender and receiver to cycle through a series of states, exchanges erroneous messages, without ever making effective progress. To check this, the state in the sender and in the receiver process that unmistakingly signify progress, were labeled as a ‘‘progress states.’’ (In fact, either one by itself would suffice.)

We should now be able to demonstrate the absence of infinite execution cycles that do not pass through any of these progress states. We cannot use the same executable from the last run, but it’s easy to setup the verifier for non±progress cycle detection:

```
$ cc ±DNP ±o pan pan.c
$ pan ±l
pan: non±progress cycle (at depth 6)
pan: wrote pan.trail
full statespace search for:
assertion violations and non±progress loops
search was not completed
vector 44 byte, depth reached 8, loops: 1
      12 states, stored
       1 states, linked
       0 states, matched    total: 13
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/1)
```

There is at least one non±progress cycle. The first one encountered is dumped into the error trail by the verifier, and we can inspect it. The results are shown in the first half of Figure 2. The channel can distort or lose the message infinitely often; true, but not too exciting as an error scenario. To see how many non±

progress cycles there are, we can use the ±c flag. If we set its numeric argument to zero, only a total count of all errors will be printed.

```
$ pan ±l ±c0
full statespace search for:
assertion violations and non±progress loops
vector 44 byte, depth reached 137, loops: 92
     671 states, stored
       2 states, linked
     521 states, matched     total: 1194
hash conflicts: 39 (resolved)
(size 2^18 states, stack frames: 0/26)
```

There are 92 cases to consider, and we could look at each one, using the ±c option (±c1, ±c2, ±c3, ...etc.) But, we can make the job a little easier by at least filtering out the errors caused by infinite message loss. We label all loss events (lines 13, 43, and 48) as progress states, using label names with the common 8± character prefix ‘‘progress,’’ and look at the cycles that remain. (Labels go behind the ‘‘::’’ flags.)

```
$ spin ±a ABP1
$ cc ±DNP ±o pan.c
$ pan ±l
pan: non±progress cycle (at depth 133)
pan: wrote pan.trail
full statespace search for:
assertion violations and non±progress loops
search was not completed
vector 44 byte, depth reached 136, loops: 1
     148 states, stored
       2 states, linked
       2 states, matched     total: 152
hash conflicts: 0 (resolved)
(size 2^18 states, stack frames: 0/26)
```

This time, the trace reveals an honest and a serious bug in the protocol. The second half of Figure 2 shows thetrace±back.

```
$ spin ±t ±r ±s ABP0
<<<<<START OF CYCLE>>>>>
proc  1 (sender)   line  13, Send err,0,0 ±> queue 2 (out)
proc  2 (receiver) line  55, Recv err,0,0 <± queue 2 (in)
proc  2 (receiver) line  56, Send nak,0,0 ±> queue 1 (out)
proc  1 (sender)   line  19, Recv nak,0,0 <± queue 1 (in)
spin: trail ends after 12 steps
step 12, #processes: 3
   _p[0] = 6
proc  2 (receiver)   line 36 (state 21)
proc  1 (sender) line 11 (state 6)
proc  0 (_init)  line 67 (state 4)
3 processes created
$
$ spin ±t ±r ±s ABP1
...
proc  2 (receiver) line  39, Recv mesg,0,0 <± queue 2 (in)
proc  2 (receiver) line  47, Send err,0,0  ±> queue 1 (out)
proc  1 (sender)   line  20, Recv err,1,0  <± queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 ±> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <± queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  ±> queue 1 (out)
proc  1 (sender)   line  21, Recv nak,0,0  <± queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 ±> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <± queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  ±> queue 1 (out)
<<<<<START OF CYCLE>>>>>
proc  1 (sender)   line  21, Recv nak,0,0  <± queue 1 (in)
proc  1 (sender)   line  12, Send mesg,0,0 ±> queue 2 (out)
proc  2 (receiver) line  39, Recv mesg,0,0 <± queue 2 (in)
proc  2 (receiver) line  52, Send nak,0,0  ±> queue 1 (out)
spin: trail ends after 226 steps
...
```

Figure 2. Error Trails ± Extended Alternating Bit Protocol

After a single positive acknowledgement is distorted and transformed into an `err` message, sender and receiver get caught in an infinite cycle, where the sender will stubbornly repeat the last message for which it did not receive an acknowledgement, and the receiver, just as stubbornly, will reject that message with a negative acknowledgment.

Digging Deeper

This manual can only give an outline of the main features of *spin*, and the more common ways in which it can be used for verifications. There is a number of *spin* features that have not been discussed here, but that may be useful for tackling verification problems.

Spin also allows for a straightforward verification of 'tasks,' or 'requirements,' modeled as *never*±claims. That is, if the user formalizes a task that is claimed to be performed by the system, *spin* can quickly either prove or disprove that claim. A never±claim is equivalent to a Büchi Automaton, and can thus model any linear time temporal logic formula. In the newer versions of *spin*, the user can use normal LTL syntax to define the requirements, and let *spin* perform the translation chore to the corresponding never±claim.

Spin also allows the user to formalize 'reductions' of the system state space, which can be used to restrict a search it to a user defined subset (again, using never claims, this time to 'prune' the statespace). With this method it becomes trivial to verify quickly whether or not a given error pattern is within the range of behaviors of a system, even when a complete verification is considered to be infeasible.

For details about these alternative uses of PROMELA and the *spin* software, refer to [5]. The extensions to *spin* with version 2.0 are outlined in a manual named `Doc/WhatsNew.ps` in the standard distribution, and will be more fully documented in a new set of coursenotes on 'The Verification of Concurrent Systems,' that is scheduled to be published in 1997.