AT&T Bell Laboratories
Murray Hill, NJ 07974

Computing Science Technical Report No. 157

Tutorial: Design and Validation of Protocols

Gerard J. Holzmann

May 1991

Tutorial: Design and Validation of Protocols

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

It can be remarkably hard to design a good communications protocol, much harder even than it is to write a normal sequential program. Unfortunately, when the design of a new protocol is complete, we usually have little trouble convincing *ourselves* that it is trivially correct. It can be a unreasonably hard to prove those facts formally and to convince also others. Faced with that dilemma, a designer usually decides to trust his or her instincts and forgo the formal proofs. The subtle logical flaws in a design thus get a chance to hide, and inevitably find the worst possible moment in the lifetime of the protocol to reveal themselves.

Though few will admit it, most people design protocols by trial and error. There is a known set of trusted protocol standards, whose descriptions are faithfully copied in most textbooks, but there is little understanding of why some designs are correct and why others are not. To design and to analyze protocols you need tools. Until recently the right tools were simply not generally available. But that has changed. In this tutorial we introduce a state-of-the-art tool called SPIN and a specification language called PROMELA, and we show how these can be used to design reliable protocols.

May 1991

Tutorial: Design and Validation of Protocols

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, NJ 07974

1. INTRODUCTION

> ''The queen's *levée* took a similar course to that of the king. The maid of honour had the right to pass the queen her chemise. The lady in waiting helped her put on her petticoat and dress. But if a princess of the royal family happened to be present, she had the right to put the chemise on the queen. On one occasion the queen had just been undressed by her ladies. Her chambermaid was holding the chemise and had just presented it to the maid of honour when the Duchess of Orleans came in. The maid of honour gave it back to the chambermaid who was about to pass it to the duchess when the higher-ranking Countess of Provence entered. The chemise now made its way back to the chambermaid, and the queen finally received it from the hands of the countess. She had had to stand the whole time in a state of nature, watching the ladies complimenting each other with her chemise.''
>
> *Court Society* -- Norbert Elias, Pantheon Books, 1983

This anecdote, quoted from an account by a chambermaid of Marie-Antoinette, nicely illustrates what happens if a presumably reasonable set of rules is interpreted rigidly in an unusual situation. The goal of the protocol of court etiquette, which ultimately was to please the queen, was certainly undermined by the strict enforcement of the rules. The link with computer protocol design is quickly made. After all, the rigid interpretation of a fixed set of rules is something in which computers excel. As at the French court, this works fine in the expected cases, but can backfire when an unexpected sequence of events occurs.

If the goal of protocol design can be summed up into one phrase it should be that the designer has the difficult task to deal with events that are, at the time of design, partly unpredictable. ''Expect the unexpected.'' In a remarkably astute assessment of the flaws in an early tele-communication system, based on fire-signals with predefined meanings, the historian Polybius wrote in the 2nd Century B.C.:

> ''.... it is chiefly unexpected occurrences which require instant consideration and help — all such matters defy communication by fire signal. For it is quite impossible to have a preconcerted code for things which there is no means of foretelling.''
>
> *The Histories* -- Polybius, Book X, Chapter 43.

In Polybius' days, the problem was to find a good method for encoding and communicating an unexpected event to a remote peer. Of course, during a communication further unexpected events can take place. The remote peer may miss messages repeatedly, or may try to initiate an urgent communication at the same time that we do. Design errors in a protocol typically hide in scenarios like these. These scenarios are so unlikely that no designer in his right mind will consider them. The first hard-learned lesson in protocol design is, however, that unlikely events really do happen and have to be dealt with.

This leads to the curious observation that the consequences of an error are often far more important that the probability of the error. Unlikely events cannot be ignored simply because they have a low probability of occurring. The worst kinds of errors are precisely the low-probability errors with grave consequences. Such errors reliably escape random testing and land in our implementations, waiting patiently for the wrong moment to strike. An example is the hypothetical once-in-a-lifetime error that can paralyze the entire U.S. national telephone network for the better part of a day. Alas, since January 15, 1990 we know that the word 'hypothetical' can be omitted from that sentence.

To tackle the protocol design problem we need a rigorous design discipline and a method to study the correctness of our solutions. To prove the essential properties of our design we have to prove, preferably mechanically, that there is no scenario that can destroy them. In this tutorial we will explore how this can be done.

In Section 2 we discuss five essential elements of a protocol specification. In Section 3 we introduce the concept of a protocol validation model. A validation model is an abstraction of a design decision and a prototype of an implementation. In Section 4 we show how correctness requirements can be expressed in the language, PROMELA, that we will use for specifying validation models. In Section 5 we introduce an automated tool called SPIN for mechanically verifying the validity of correctness requirements, and give some examples of its application. Section 6 discusses the application of SPIN to large problems. Appendices A and B summarize the main language features of PROMELA. The table below gives an overview of the main sections of the paper.

2. THE BASIC ELEMENTS OF A PROTOCOL

It is fairly difficult to give a strictly formal and unambiguous definition of any given abstract function in plain English. Protocol definitions are no exception, and even the formal language of an international standard often relies on the good-will and common sense of the reader, and can leave much room for misinterpretation.

An Informal Example

Below is a simple example of how protocols are typically specified. The example is taken from a paper published by Lynch in 1968 [Lynch '68]. The paper inspired the publication of the well-known paper introducing the alternating bit protocol [Bartlett et al. '69]. The line-numbers are added for reference.

```
 1   The protocol defines a simplex data-transfer  channel,  i.e.,  it
 2   will  pass  data  in only one direction, from sender to receiver.
 3   Control information, however, flows in both  directions.   It  is
 4   assumed that the system has perfect error detection.
 5   To each message sent from A to B we attach an  extra  bit  called
 6   the  alternation bit.  After B receives the message it decides if
 7   the  message  is  error-free.   It  then  sends  back  to   A   a
 8   verification   message,   consisting  of  a  single  verify  bit,
 9   indicating whether or  not  the  immediately  preceding  A  to  B
10   message  was error-free.  After A receives this verification, one
11   of three possibilities hold:
12           1. The A to B message was good
13           2. The A to B message was bad
14           3. A cannot tell if the A to B message was  good  or  bad
15   because the verification message (sent from B to A) was in error
16   In cases 2 and 3 A resends the same A to B message as before.  In
17   case  1  A  fetches  the  next  message to be sent, and sends it,
18   inverting the setting of the alternation bit with respect to  the
19   previous A to B message.
20   Whenever B receives a message that is not in  error  it  compares
21   the alternation bit of this new message to the alternation bit of
22   the most recent error-free reception.  If  the  alternation  bits
23   are  equal  the  new message is not accepted.  The new message is
24   accepted  only  if  the  two  alternation   bits   differ.    The
25   verification  messages  from B to A indicate error-free reception
26   independently of the acceptance of the messages.
27   Initialization of this scheme depends upon A and B agreeing on an
28   initial  setting of the alternation bit.  This is accomplished by
29   an A to B message whose error-free reception (but not necessarily
30   acceptance)  forces B's setting of the alternation bit.  Multiple
```

```
receptions of such a message cannot do harm.
This protocol has the property that every message fetched by A is
received error-free at least once and accepted at most once by B.
```

The description certainly looks reasonable and implementable. For one thing, it is much clearer than the description of essentially the same protocol in [Schwartz ’63]. But, is this example protocol fully specified? Could we determine if the protocol is correct before we go ahead and implement it? Before we address these points, let us first try to answer a more basic question: what precisely should a protocol define?

What Is A Protocol?

Figure 1 shows two communicating entities, A and B. There is an upper interface *(1)* to code that uses the A-B protocol, and a lower interface *(2)*, to code that implements the A-B protocol. The lower interface *(2)* can be thought of as being implemented at another level of abstraction by the dotted protocol layer from Figure 1, based on a still lower-level interface *(3)*. At the lowest level of abstraction interface (3) should match the specification of the target physical interface. At each level of abstraction, the upper interface becomes an idealized version of the lower interface, by hiding details (imperfections) and providing higher-level functions. What we call a ‘protocol’ is what we see if we cut the hierarchy at one specific level of abstraction and look at the interface that is then exposed. The A-B protocol in Figure 1, for instance, is a cut of the protocol stack at interface (1).

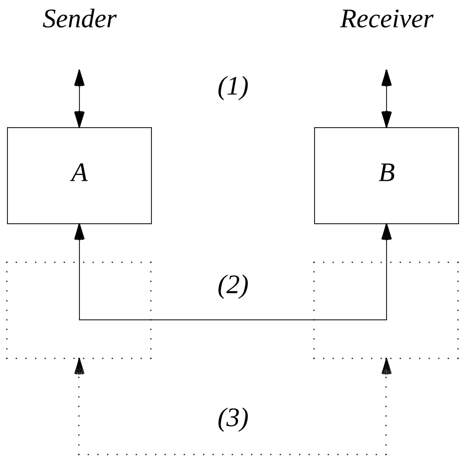

Figure 1 — Levels of Abstraction

In a typical protocol design problem, we are given the characteristics of a specific interface (e.g. (3)), and we are given the definition of an abstract interface (e.g. (1)). Interface (3) can define anything from the behavior of an optical fiber, to the packet layer of an OSI protocol. Interface (1), similarly, can define anything from an error-free file transfer to a remote data-base search service. The designer’s job is to transform (3) step by step until the target (1) is matched, or vice versa. To define any one step in this hierarchy, we must specify explicitly what the lower interface looks like, and how it is transformed into the upper interface.

The lower interface definition together with the specification of the process that transforms it into the upper interface, has many of the properties of a ‘language.’ The vocabulary of that language is the set of messages that can be exchanged across the interfaces. The syntax rules define the format for each such message and the grammar rules define how the messages are used in the interactions across the interfaces. In protocol jargon, the grammar rules are usually called ‘procedure rules.’ They are most easily expressed as the behavior of an abstract protocol machine that sends and receives messages.

To define a protocol, then, is to define a language. Omitting the specification of any of the language elements from a protocol specification unavoidably leads to an incomplete or ambiguous specification.

The Five Basic Elements

We can now try to make an explicit list of the basic types of elements that must be part of a complete protocol specification. It must include:

1. The *service* to be provided by the protocol (formalizing the upper interface)
2. The *constraints* of the environment in which the protocol is executed (formalizing the lower interface)
3. The *vocabulary* of messages that is used to implement the protocol
4. The *encoding* (syntax/format) of each message in the vocabulary
5. The *procedure rules* guarding message exchanges (grammar/behavior)

The core of the protocol definition is **5**, the procedure rules. A correctness claim is typically a claim about the possibility or impossibility of a particular behavior, and it is therefore especially important that we have good formalisms for expressing and for verifying process behaviors. As an exercise, we can try to identify these five basic elements in the example protocol of Lynch.

Lines 1 to 4 tersely define the upper and lower interface. The service to be provided through the upper layer is reliable simplex data transfer. The service assumed to be available via the lower layer (the constraint) is a channel that can lose, distort, or transfer messages, with perfect classification of the distorted and correctly transferred messages.

Message format is mentioned in lines 5 to 10, amidst a discussion of procedure rules. Lines 10 to 31, discuss procedures rules alone, and lines 32 and 33 contain a claim about the behavior of the protocol. Formally, this claim is not part of the protocol specification itself, but we can require that the specification include enough information to verify it. The protocol vocabulary and message formats are not explicitly specified. The procedure rules are stated in a pleasant informal tone, but without following any particular formalism. This becomes especially clear on lines 27 to 31, which contain only an idea for initialization.

The Design Problem

The heart of the protocol design problem is the design of a consistent set of procedure rules. We would like to explore a way of defining and checking these rules on a fairly high level of abstraction, i.e., before we settle the details of an implementation. We would like to defer decisions on, for instance, message format definitions, the layout of bits and fields in messages, until we have found a correct set of rules. With such a method, we could formalize the example specification in such a way that we can prove conclusively if the correctness claim on lines 32 and 33 is justified.

It can be argued that a good engineering discipline must have three characteristics. It must allow the user to

- ○ discriminate between requirements and implementations,
- ○ use engineering models (prototypes) to verify design decisions, and
- ○ predict essential characteristics of a product before it is implemented.

To allow us to design protocols in this manner, we need an unambiguous notation for expressing procedure rules and correctness claims, we need a method for building prototypes, and we need a method for mechanically verifying the soundness of our design decisions, as cast in the protocol prototypes. We explore these issues in the next few sections.

3. PROTOCOL VALIDATION MODELS

Our first task is to develop a notation for formalizing the procedure rules of a protocol in such a way that we can easily verify their completeness and logical consistency. At this level, we are not interested in a full implementation of the rules. Hence, we do not specify any specific encoding of messages, memory allocation routines, or general operating system support. The models we build are primarily meant for validation, and are therefore called *validation models*. By supplying the missing details, a validation model can be expanded into a full implementation, but we will not cover that here. We merely require that the validation model has enough detail to allow us to check its properties rigorously, but not so much detail that analysis would become intractable. The language we develop here is called PROMELA, short for Protocol Meta-Language.

The Language PROMELA

Let us look at how the lower-layer constraints of the example protocol could be specified in PROMELA. We have two processes, named A and B, communicating with each other via a lower protocol layer, as shown in Figure 2.

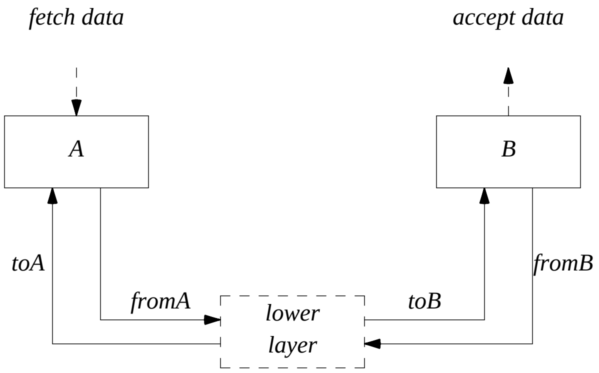

Figure 2 — A-B protocol

In this case the lower layer corresponds to a physical connection, but that is irrelevant to the validation model. The lower layer has a behavior (i.e., follows certain procedure rules) that we must formalize. We make minimal assumptions about the format of a message. Process A sends data messages to process B that consist of a data field and an alternation bit. Process B answers with control messages, containing just a single bit of information. So, in the validations we can work with two formal message types, declared in PROMELA as:

```
mtype = { data, control }                /* data and acks      */
```

Since the correct working of the protocol must be independent of the data field, we can either delete the field completely from the validation model, or, as we shall do below, use a place-holder. The place-holder we choose consists of a single byte of information. The message channels between A and B can then be formalized in PROMELA as follows.

```
chan fromA = [N] of { byte, byte, bit }; /* data, udata, seqno  */
chan toB   = [N] of { byte, byte, bit }; /* data, udata, seqno  */
chan fromB = [N] of { byte, bit };       /* control, seqno      */
chan toA   = [N] of { byte, bit };       /* control, seqno      */
```

The channels from A to B carry three unnamed fields, of which only the width is specified: a message-type field (specifying `data` or `control`), encoded liberally in one byte of information, a dummy data field, also of one byte, and the alternation bit. The channels from B to A just have the type field and the verify bit.

`N` is a constant that specifies the capacity of the channel, i.e., the maximum number of messages it can hold. For now we can guess that, for this protocol, this maximum need not be larger than one. We can check later that this assumption is justified with a formal validation.

A PROMELA specification consists of only three basic building blocks: message channels, processes and variables. We have just shown how message channels are formalized and declared (the least intuitive part of the language). The next example shows how processes and variables are formally declared, and how message channels are used.

Model of an Ideal Channel

As a first approximation, we could specify an ideal lower layer, that flawlessly shuttles messages between A and B, as follows (as usual, line numbers are added and are not part of the specification).

```
proctype lower(chan fromA, toA, fromB, toB)
{    byte d; bit b;

     do
```

```
5       :: fromA?data(d,b) -> toB!data(d,b)
6       :: fromB?control(b) -> toA!control(b)
7       od
8  }
```

We have specified a behavior for the lower protocol in a `proctype` definition. The process type is named `lower` and has four parameters, one for each message channel that it must access (line 1). The body of the process definition is enclosed in curly braces. It starts with the declaration of two internal variables on line 2. One variable is of type `byte` and one is of type `bit`. We have now seen four different types of PROMELA objects:

`proctype`, `chan`, `bit`, and `byte`

There are only two other types (both data types, i.e., for variables)

`short`, and `int`.

A object of type `bit` is a variable that can hold a single bit of information. An object of type `byte` is a variable with a type that is equivalent to a C `unsigned char`. The precise range of such a variable is machine dependent. On most machines it suffices to store 8 bits of information. Objects of types `short` and `int` are signed variables that are mapped onto the same data types in C. On most machines again, a `short` stores 16 bits of information, and an `int` stores 32 bits.

The behavior is specified as a single loop (lines 4-7). The syntax is derived from Dijkstra's guarded command language [Dijkstra '75], and Hoare's language CSP [Hoare '78], but the semantics are different (cf. Appendix A). A `do`-loop, for instance, is repeated until it is explicitly terminated by the execution of a `break` statement, or a `goto` jump.

The loop on lines 4-7 is not broken, and therefore will not terminate. There are two options in the loop, each starting with the `::` flag, and each with two statements. Each time though the cycle of the loop, one of the two options is selected for execution, using the rules stated below. The token `->` is a statement separator. In fact, PROMELA allows two statement separators, the traditional semicolon, and the arrow. They are semantically completely equivalent.

The first statement in each option is called a *guard*. The option can only be selected if the guard is executable. The guard of the first option, on line 5, specifies the reception of a message `data(d,b)` from channel `fromA`.

```
fromA?data(d,b)
```

This receive operation is executable if and only if a message of the required type is queued in channel `fromA`. Channels behave as FIFO queues. A message of type `data` must therefore be at the head of that queue. The receive statement is unexecutable when, for instance, a message of type `control` is at the head of channel `fromA`. The number of parameters specified in the receive operation must always match the number specified in the corresponding channel declaration.

The second statement in both options is a send operation. For the first option, for instance,

```
toB!data(d,b)
```

specifies the sending of message `data(d,b)` to channel `toB`. By default, the send action is only executable if the target channel is not full. This means that in validation runs it is considered a design error if message queues can be made to overflow. (The user can override the default though and stipulate that messages sent to full queues must be discarded.) ''Executability'' is a central concept in PROMELA, and the main tool for formalizing process synchronization in a validation model.

Executability in PROMELA

Every statement in PROMELA is either executable or non-executable. The semantics of PROMELA (summarized in Appendix A) specify the rules of executability for every type of statement. Assignment statements, for instance, are always executable. Boolean conditions are executable if and only if they are true. Any statement that is non-executable can block the executing process. In the loop of the example above, the state of the channels determines which of the two guards will be executable and thus selectable by the lower layer process. If no guard is executable, the process blocks. If more than one guard is executable, one of them is selected at random.

Model of a Non-Ideal Channel

The real lower layer for the example protocol does not always transfer messages correctly. In the informal description, lines 3-4, it is implicitly assumed that the channel can corrupt messages, but not lose them. The error detection scheme is assumed to be flawless (not a realistic assumption, but certainly a practical and a common one). To build our validation model we will assume that the error detection scheme will label corrupted messages appropriately as `error` messages. To formalize this, we first expand our message vocabulary to three types of messages (luckily we used a `byte` for the message-type field; enough to distinguish between up to 256 different message types):

```
mtype = { data, control, error }
```

We can now formalize the new behavior as follows.

```
proctype lower(chan fromA, toA, fromB, toB)
{   byte d; bit b;

    do
    :: fromA?data(d,b) ->
            if
            :: toB!data(d,b)        /* correct   */
            :: toB!error            /* distorted */
            fi
    :: fromB?control(b) ->
            if
            :: toA!control(b)
            :: toA!error
            fi
    od
}
```

The lower layer now has two possible responses to an incoming message. It can either forward the message correctly, as before, or it can change the message into an error message. The `if` statement specifies a number of options for execution, just like a `do` statement. Unlike the `do` statement, however, the `if` statement terminates when the option that was selected terminates, i.e., it is not repeated. The mechanism for selecting an option is the same as before. In the two `if` statements above both options consist of just a single send statement. If we wanted to model the possibility of message loss, we could add yet another option to this set, consisting of a single statement `skip`. The `skip` statement is the null operation of PROMELA. It is always executable and has no effect.

This definition of process `lower` gives a description of the behavior of the lower layer protocol that accurately matches the assumptions of the protocol designer. To complete the validation model, we must combine it with the declarations of the channels, and we must find a place where a process of type `lower` is instantiated with the appropriate channels. We return to that below, after we discuss the modeling of sender and receiver process types A and B.

Model of the Sender and Receiver Process

We start by taking a closer look at the procedure rules for the sender process A. The information we need is on lines 5-19 in the informal specification by Lynch. The code below is annotated with references to the line numbers in the informal description, where the corresponding design decisions are mentioned.

```
proctype A(chan in, out)
{   byte mt;                        /* message data            */
    bit  at;                        /* alternation bit         */
    bit  vr;                        /* verify bit              */

    FETCH;                          /* get a new mesg          */
    out!data(mt,at);                /* send it                 */
    do
    :: in?control(vr) ->            /* line 11, await response */
            if
```

```
11              :: (vr == 1) ->         /* line 12, correct send   */
12                      FETCH;          /* line 17, new message    */
13                      at = 1-at       /* line 18, toggle bit     */
14              :: (vr == 0) ->         /* line 13, send error     */
15                      skip            /* line 16, don’t fetch    */
16              fi;
17              out!data(mt,at)         /* line 16                 */
18      :: in?error(vr) ->              /* line 14-15, recv error  */
19              out!data(mt,at)         /* line 16                 */
20      od
21  }
```

The only new language features that we have used is the assignment to toggle the alternation bit, and the conditions as guards in the `if` statement. The parameter `vr` on line 18 is not used, but must be present to fulfill the requirement that the number of parameters in a receive equal the number of message fields declared for the corresponding channel. Assignments are defined to be always executable, unconditionally. The evaluation of the right-hand side of an assignment can have no side-effects. (This is not true in C.) A condition, or in general any expression that is used as a statement, is only executable if it evaluates to a non-zero value. Expressions used as statements must be ‘pure,’ that is they may not have side-effects when evaluated. A trivial encoding for the dummy statement `skip` is therefore the expression `(1)`. It has no effect, and is always executable.

We still have to decide on the modeling of the initialization idea from lines 27-31 in Lynch’s informal specification, and we have to expand the macro `FETCH` somehow. But, before we do that, let us first look at the matching receiver.

```
proctype B(chan in, out)
{   byte mr;                        /* message data              */
    byte last_mr;                   /* mr of last error-free msg */
    bit  ar;                        /* alternation bit           */
    bit lar;                        /* ar of last error-free msg */

    do
    :: in?error(mr,ar) ->           /* lines 7-10    */
            out!control(0)          /* lines 8,25,26 */
    :: in?data(mr,ar) ->            /* lines 7-10    */
            out!control(1);         /* lines 8,25,26 */
            if                      /* line 20       */
            :: (ar == lar) ->       /* line 21       */
                    skip            /* line 23       */
            :: (ar != lar) ->       /* line 24       */
                    ACCEPT;         /* line 24       */
                    lar = ar;       /* line 22       */
                    last_mr = mr
            fi
    od
}
```

We introduced a new macro `ACCEPT` that, just like the previous one `FETCH` remains to be expanded, but apart from these details (and the proper initializations) the behavior specifications are now complete. A trivial implementation is to use the macro `FETCH` to obtain a sequence of integers, modulo some maximum number `MAX`

```
#define FETCH   mt = (mt+1)%MAX
```

If we let the receiver remember the last number received, we can build in a simple check to verify that the same message cannot be accepted twice in a row by `B` (line 33 from the informal specification):

```
#define ACCEPT  assert(mr == (last_mr+1)%MAX)
```

The statement `assert(e)` is a pre-defined statement in PROMELA, with `e` an arbitrary expression. The statement is always executable and has no effect. It is an error if expression `e` can be false when the assertion is executed. We can use an automated validator to prove that this is impossible.

The assertion above does not cover the first part of the correctness claim on lines 31-33 in the specification.

We will see later how this requirement can be expressed and checked. First, we describe how the process behaviors can be included into a complete validation model.

The Initial PROMELA Process

A `proctype` definition only defines process behavior, it does not specify when that behavior must be performed, or how it is to be instantiated. For this, every validation model is defined to have an initial process, that performs much the same function as the `main` routine of a C-program. The minimal PROMELA model does not declare any objects or instantiate any processes:

```
init { skip }
```

More interesting is an `init` process that declares the channels from Figure 2 and instantiates a single copy of each of the `proctypes` `A`, `B`, and `lower`.

```
 1  #define N   2
 2  #define MAX 8
 3  #define FETCH       mt = (mt+1)%MAX
 4  #define ACCEPT      assert(mr == (last_mr+1)%MAX)
 5
 6  mtype = { data, control, error };
 7
 8  #include "lynch0.A"
 9  #include "lynch0.B"
10  #include "lynch0.C"
11
12  init {
13      chan fromA = [N] of { byte, byte, bit };
14      chan toB   = [N] of { byte, byte, bit };
15      chan fromB = [N] of { byte, bit };
16      chan toA   = [N] of { byte, bit };
17
18      atomic {
19              run A(toA, fromA);
20              run B(toB, fromB);
21              run lower(fromA, toA, fromB, toB)
22      }
23  }
```

The three processes are initiated in an `atomic` statement, to guarantee that they all start at the same time. Note, there is no semicolon at the end of the atomic block: semi-colons are used as statement separators, not as statement terminators.

Simulation and Validation with SPIN

We have not yet formalized the complete correctness requirement from the informal specification, and we have not yet included the proper protocol initialization, but just as an experiment we can try to check the behavior of the model as it stands. In the example simulation-runs below we assume a UNIX® System environment, with '$' the shell prompt:

```
$ spin lynch0
spin: "./lynch0.B" line 16: assertion violated
#processes: 4
                _p = 8
proc  3 (lower) line 11 (state 9)
proc  2 (B)     line 16 (state 8)
proc  1 (A)     line 8 (state 14)
proc  0 (_init) line 23 (state 5)
4 processes created
$
```

The default usage of SPIN starts a random simulation run of the validation model. Almost no correctness checks are performed during SPIN simulations. The exception are inescapable errors, such as system deadlock, unspecified reception, and assertion violation. (This excludes a larger class of correctness criteria that we discuss later.) In the above case the simulation run aborted when an assertion violation occurred.

Before SPIN exits it prints the control-flow state of all processes at the time the error was detected.

To find out what happened we need more information about the run itself. First, we need to make sure that we have a reproducible error. (After all, this is a concurrent system with several asynchronously executing processes, there are many possible execution sequences, all equally likely.) We can secure a reproducible run by selecting a fixed seed for the random number generation (by default SPIN uses the UNIX System's `time` command as a seed). We can do this as follows:

```
$ spin -n123 lynch0
spin: "./lynch0.B" line 16: assertion violated
#processes: 4
                _p = 8
proc  3 (lower) line 11 (state 9)
proc  2 (B)     line 16 (state 8)
proc  1 (A)     line 8 (state 14)
proc  0 (_init) line 23 (state 5)
4 processes created
$
```

The seed selected was `123`, the same error was hit, so we can now repeat the run reliably, by using the same seed consistently in all further probings into the nature of this error.

SPIN allows us to get arbitrarily detailed information about the simulation run leading into the assertion violation, by combining different option flags. A quick overview of the available information can be obtained by providing an illegal option flag such as "`spin -h`" or "`spin -?`." Five of these options are relevant here:

```
$ spin -?
        -g print all global variables
        -l print all local variables
        -p print all statements
        -r print receive events
        -s print send events
```

So we can try:

```
$ spin -n123 -r lynch0
proc  3 (lower) line   5, Recv data,1,0  <- queue 2 (fromA)
proc  2 (B)     line   8, Recv error,0,0 <- queue 3 (in)
proc  3 (lower) line  10, Recv control,0 <- queue 4 (fromB)
proc  1 (A)     line   9, Recv control,0 <- queue 1 (in)
proc  3 (lower) line   5, Recv data,1,0  <- queue 2 (fromA)
proc  2 (B)     line  10, Recv data,1,0  <- queue 3 (in)
proc  3 (lower) line  10, Recv control,1 <- queue 4 (fromB)
proc  1 (A)     line   9, Recv control,1 <- queue 1 (in)
proc  3 (lower) line   5, Recv data,2,1  <- queue 2 (fromA)
proc  2 (B)     line  10, Recv data,2,1  <- queue 3 (in)
proc  3 (lower) line  10, Recv control,1 <- queue 4 (fromB)
spin: "./lynch0.B" line 16: assertion violated
#processes: 4
                _p = 8
proc  3 (lower) line 11 (state 9)
proc  2 (B)     line 16 (state 8)
proc  1 (A)     line 8 (state 14)
proc  0 (_init) line 23 (state 5)
4 processes created
```

Trivially, we can also filter out uninteresting data, for instance the actions of the lower layer channel process:

```
$ spin -n123 -r lynch0 | grep -v lower
proc  2 (B)     line   8, Recv error,0,0 <- queue 3 (in)
proc  1 (A)     line   9, Recv control,0 <- queue 1 (in)
proc  2 (B)     line  10, Recv data,1,0  <- queue 3 (in)
proc  1 (A)     line   9, Recv control,1 <- queue 1 (in)
proc  2 (B)     line  10, Recv data,2,1  <- queue 3 (in)
spin: "./lynch0.B" line 16: assertion violated
```

```
#processes: 4
                _p = 8
proc  2 (B)     line 16 (state 8)
proc  1 (A)     line 8 (state 14)
proc  0 (_init) line 23 (state 5)
4 processes created
```

Or, we can add explicit `printf` statements to the PROMELA source text for the model to keep track of what is happening. For instance, we can experiment by expanding the `ACCEPT` macro,

```
#define ACCEPT  printf("ACCEPT %d0, mr); assert(mr == (last_mr+1)%MAX)
```

which produces:

```
pipe: spin -n123 -r lynch0 | grep -v lower
proc  2 (B)     line   8, Recv error,0,0 <- queue 3 (in)
proc  1 (A)     line   9, Recv control,0 <- queue 1 (in)
proc  2 (B)     line  10, Recv data,1,0  <- queue 3 (in)
proc  1 (A)     line   9, Recv control,1 <- queue 1 (in)
proc  2 (B)     line  10, Recv data,2,1  <- queue 3 (in)
ACCEPT 2
spin: "./lynch0.B" line 16: assertion violated
#processes: 4
                _p = 9
proc  2 (B)     line 16 (state 9)
proc  1 (A)     line 8 (state 14)
proc  0 (_init) line 23 (state 5)
4 processes created
```

The error is now clear. The first message accepted carries the integer `2`. The message with integer value `1` is received correctly but not accepted by process `B`. The only reason for that is line 13 in `proctype B`: the alternation bit on the first message equals the value of `lar`. This initialization problem can be patched (we didn't look seriously yet at the rules for the initialization that are given in the informal specification) by giving `lar` a non-zero initial value (by default everything in PROMELA has initial value zero):

```
 1  proctype B(chan in, out)
 2  {   byte mr;                        /* message data              */
 3      byte last_mr;                   /* mr of last error-free msg */
 4      bit  ar;                        /* alternation bit           */
 5      bit lar=1;                      /* ar of last error-free msg */
        ....
21  }
```

With this modification, the simulation proceeds as expected.

```
$ spin lynch0
ACCEPT 1
ACCEPT 2
ACCEPT 3
ACCEPT 4
ACCEPT 5
ACCEPT 6
ACCEPT 7
ACCEPT 0
ACCEPT 1
ACCEPT 2
ACCEPT 3
...
```

repeating the series `ACCEPT 0` to `ACCEPT 7` ad infinitum. Of course, we cannot prove with simulation alone that the assertions cannot be violated. Simulation can only serve as a quick debugging tool, not for validation. An exhaustive validation, *proving* that the assertions can never be validated in this validation model, is performed in three steps:

1. Generate the analyzer source, using SPIN option `-a`.
2. Compile this source with the C-compiler.
3. Run the resulting executable analyzer.

This analysis proceeds as follows (after removing the `printf` statement we added earlier for debugging).

```
$ spin -a lynch0        # generate analyzer
$ cc -o pan pan.c       # compiler source in pan.c
$ pan                   # run
full statespace search for:
        assertion violations and invalid endstates
vector 84 byte, depth reached 350, errors: 0
    1558 states, stored
       2 states, linked
    1169 states, matched           total:     2729
hash conflicts: 201 (resolved)
(max size 2^18 states, stackframes: 0/102)
unreached in proctype _init:
        reached all 5 states
unreached in proctype lower:
        line 16 (state 14)
        reached: 13 of 14 states
unreached in proctype B:
        line 21 (state 17)
        reached: 16 of 17 states
unreached in proctype A:
        line 21 (state 17)
        reached: 16 of 17 states
0.4u 0.3s 5r     pan
```

The first line in the printout tells us what type of analysis was performed. In this example it was a full statespace search for assertion violations and invalid endstates, such as deadlocks (we will discuss the other possible types of analyses later).

The next six lines report some statistics on the analysis itself that will become clear later. For now, just note that the longest unique execution sequence that was analyzed contained 350 statements (depth reached), and a total of 1558 unique system states were encountered during the search, each occupying 84 bytes of storage.

The remainder of the printout gives us information about the statements in the validation model that were analyzed during the search. Unreachable states in this listing either necessarily indicate dead code in the model for a full statespace search that is run to completion. (Later, when we look at partial searches of large models, the unreachable states give information about the coverage of the search.) The unreached states in the current listing are benign: they point at the end-states of the three processes that we intended not to terminate.

Modeling Protocol Initialization

We haven't looked seriously at the rules for the proper initialization of Lynch's protocol yet. The informal specification is rather vague:

```
27   Initialization of this scheme depends upon A and B agreeing on an
28   initial  setting of the alternation bit.  This is accomplished by
29   an A to B message whose error-free reception (but not necessarily
30   acceptance)  forces B's setting of the alternation bit.  Multiple
31   receptions of such a message cannot do harm.
```

One way to implement this is to change the start of `proctype A` into:

```
  1 proctype A(chan in, out)
  2 {   byte mt;                        /* message data    */
  3     bit  at;                        /* alternation bit */
  4     bit  vr;                        /* verify bit      */
  5
> 6     out!data(0,1);                  /* initialize      */
> 7     do
> 8     :: in?error(vr) ->              /* ack error       */
> 9             out!data(0,1)           /* repeat          */
>10     :: in?control(0) ->             /* send error      */
>11             out!data(0,1)           /* repeat          */
```

```
>12     :: in?control(1) ->             /* success: done   */
>13             break
>14     od;
 15     FETCH;
        ...
 30 }
```

The corresponding changes in `proctype B` below refer to lines 27-31 in the informal specification:

```
  1 proctype B(chan in, out)
  2 {   byte mr;                        /* message data              */
  3     byte last_mr;                   /* mr of last error-free msg */
  4     bit  ar;                        /* alternation bit           */
  5     bit lar=1;                      /* ar of last error-free msg */
> 6     bit ini;                        /* lines 27-31   */
  7
  8     do
  9     :: in?error(mr,ar) ->           /* lines 7-10    */
 10             out!control(0)          /* lines 8,25,26 */
 11     :: in?data(mr,ar) ->            /* lines 7-10    */
 12             out!control(1);         /* lines 8,25,26 */
>13             if                      /* lines 27-31   */
>14             :: (!ini) ->            /* lines 27-31   */
>15                     ini = 1;        /* lines 27-31   */
>16                     lar = ar        /* lines 27-31   */
>17             :: ini ->               /* lines 27-31   */
 18                     if                /* line 20     */
 19                     :: (ar == lar) -> /* line 21     */
 20                             skip      /* line 23     */
 21                     :: (ar != lar) -> /* line 24     */
 22                             ACCEPT;   /* line 24     */
 23                             lar = ar; /* line 22     */
 24                             last_mr = mr
 25                     fi
 26             fi                      /* lines 27-31   */
 27     od
 28 }
```

It is quickly confirmed by simulation and validation that the protocol works correctly, this time independent of the initial value of variable `lar` in `proctype B`.

4. EXPRESSING CORRECTNESS REQUIREMENTS

There are no ‘‘correct’’ protocols. Protocols can only be called correct with respect to specific correctness requirements. Usually, in protocol validations we check a protocol for a set of fairly standard requirements, such as absence of deadlocks (i.e., improper terminations), and for a range of soundness and completeness requirements, such as absence of unspecified receptions, dead code segments, race conditions and buffer overrun.

To prove that a design effectively solves a given problem, such as data transfer, however, requires us to express more specific types of correctness requirements. These requirements typically state that a certain behavior of the protocol system is either feasible or infeasible. In this section we discuss how such requirements can be expressed in PROMELA and checked with SPIN.

Formalizing Behavior

The behavior of a validation model is completely defined by the set of execution sequences that it defines, including all possible interleavings of the concurrent behavior of its processes. An execution sequence is a finite, ordered set of system states, and a state is a complete specification of values of local and global variables, control flow points of running processes, and contents of message channels. A validation model can reach a given state by executing PROMELA statements, using the semantics of executability discussed earlier.

PROMELA validation models are by definition finite (all ranges are bounded, including the maximum number of processes and message channels that can be created). There is therefore necessarily a bounded, and enumerable, number of execution sequences that defines the behavior of any given PROMELA model. We can distinguish two different types of sequences:

- ○ Terminating sequences.
- ○ Cyclic sequences.

The specification of a correctness requirement is defined minimally by propositions (Boolean expressions) on system states. The expressions can in principle refer to all the elements of a system state: control-flow points of processes, local and global data, and message queues. Formally, each proposition defines a mapping of all reachable system states onto the Boolean values *true* and *false*.

A simple example of the use of these propositions to define a correctness requirement is the PROMELA assertion statement that we used in the definition of the `ACCEPT` macro earlier. An assertion specifies a proposition that must always be true when a given process can reach a specific control-flow point (i.e., the place in the process body where the assertion is placed). In this case, the proposition only depends on the control-flow point of a single process, and on the relative values of two local variables within that process.

If more than one simple proposition is used, possibly applying to more than one process, more complicated correctness requirements can be build by specifying, for instance, the temporal order in which the propositions are required to hold. Alternatively, and equivalently, we may specify which temporal orderings are forbidden. (In fact, only the second flavor of temporal orderings is used in PROMELA.)

There are three ways in which correctness criteria can be expressed in PROMELA

- ○ Assert statements, as discussed above.
- ○ Three types of validation labels.
- ○ Temporal claims, specifying invalid system behavior.

We will illustrate the use of each type of correctness requirement with examples below.

4.1. ASSERTIONS

The PROMELA statement

```
assert(condition)
```

is always executable and can be placed anywhere in a PROMELA model. The condition can be an arbitrary Boolean expression. If the condition is true, the statement has no effect. The validity of the statement is violated, however, if there is at least one execution sequence in which the condition is false when the `assert` statement becomes executable.

Process Invariants

A first method to use assertions is to formalize a correctness requirement that is local to a process. We can call such a requirement a ‘‘process invariant.’’ Consider the following example

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

We could try to claim that when a process of type `A()` completes the value of variable `state` must be 2, and when a process of type `B()` completes it must be 0. This could be expressed with two process invariants as follows.

```
byte state = 1;
proctype A()
{       (state == 1) -> state = state + 1;
        assert(state == 2)
}
proctype B()
{       (state == 1) -> state = state – 1;
        assert(state == 0)
}
init { run A(); run B() }
```

The claims are false, and an automated validator could easily produce a scenario to prove it.

System Invariants

A more general application of the `assert` statement is to formalize system invariants, i.e., Boolean conditions that, if true in the initial system state, remain true in *all* reachable system states, independently of the execution sequence that leads to each specific state. To express this in PROMELA, it suffices to place the system invariant by itself in a separate, ‘‘monitor’’ process

```
proctype monitor() { assert(invariant) }
```

Once an instance of the process type `monitor` has been started (the name is irrelevant), with a regular `run` statement, it executes independently of the rest of the system. It can decide to evaluate the assertion at any time; its `assert` statement can become executable in every reachable state of the system. An exhaustive check of correctness for the system should be able to determine if any scenario exists where the monitor process executes the invariant just when it happens to be false.

Question: can we remove the ACCEPT macro from our earlier PROMELA specification from the receiver process and place it into a separate monitor process? If yes, show how; if no, why not? □

4.2. VALIDATION LABELS

End State Labels

In a finite state system, all execution sequences either terminate after a finite number of state transitions, or they cycle back to a previously visited state. Not all terminating sequences, however, are necessarily bad. In order to define what an invalid end-state in a PROMELA model is, we must be able to distinguish the expected, or *valid*, end-states from the unexpected, or invalid, ones. The term ‘‘invalid end-state’’ includes deadlock states, but also many error states that are the result of a logical incompleteness of the protocol specification. The classic example of the latter is the *unspecified reception*.

By default, the final state in a terminating execution sequence must satisfy the following two criteria to be considered a valid end-state:

○ Every process that was instantiated has reached the end of its code.
○ All message channels are empty.

But not all processes necessarily reach the end of their code. It can be perfectly valid, for instance, for server processes to enter a wait state, ready to spring back into action, after a transaction is completed, immune to the fact that all user processes have terminated. We must be able, therefore, to identify individual process states as valid end-states. (Note that by default only the end of the program block in a `proctype` definition, i.e., the closing curly brace, is a valid end-state.) This can be done with *end-state* labels. The following example, for instance, specifies a binary semaphore, that accepts `P` and `V` messages in strict alternation via a synchronous channel `sema.`

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

proctype dijkstra()
{
end:    do
        :: sema!p -> sema?v
        od
}
```

The process is non-terminating, but it should always be in its initial state, at the start of the loop, when a system execution sequence terminates. If there is more than one valid end-state within a single `proctype` definition, all label-names must still be unique. An end-state label is defined to be any label-name that has a three-character prefix `end`. So it is valid to use variations such as `enddne`, `end0`, `end_war`.

Progress State Labels

The analyzer SPIN can be asked to find all invalid cyclic execution sequences. The question then is, of course, what makes a cyclic execution sequence either valid or invalid. An invalid cyclic execution sequence is defined to be a finite sequence of statements that can be repeated infinitely often, without achieving any ‘‘progress’’ in the protocol. The user can specify precisely which statement(s) in the specification constitute progress. An example of such a statement can be the increment of a sequence number, the acceptance of a newly received message, etc. All such states are labeled with the word ‘‘progress,’’ just like the end-state labels before.

In the semaphore example we can recognize the successful passing of a semaphore test as ‘‘progress.’’ Simply by marking it as a progress state we can express the correctness criterion that the passing of the semaphore guard cannot be postponed infinitely long, e.g., by an infinite execution cycle (in the remainder of the system) that does not pass the progress state of any process of type `dijkstra`.

```
proctype dijkstra()
{
end:    do
        :: sema!p ->
progress:       sema?v
        od
}
```

If more than one state carries a progress-state label, variations with a common prefix are again valid: `progress0`, `progressisslow`, and so on. If more progress states are defined, they all carry the same weight, that is, passing any one of them will be considered progress.

Question: How could we use progress labels to show that several statements, potentially divided over more than one process, are all executed as part of the same cycle? [Hint: consider using an extra monitor process and extra messages.] □

Acceptance State Labels

Suppose we wanted to express the opposite of a progress condition, e.g., we want to formalize that something *cannot* happen infinitely often. We can express properties like this with acceptance-state labels. An *acceptance-state label* is any label starting with the character sequence ‘‘accept.’’ It marks a state that *may not* be part of a sequence of states that can be repeated infinitely often.

For example, if we replace the progress-state label in `proctype dijkstra()` with an acceptance-state label

```
proctype dijkstra()
{
end:    do
        :: sema!p ->
accept:         sema?v
        od
}
```

we claim that it is impossible to cycle through a series of `p` and `v` operations. (The claim, of course, is violated for all correct implementations of the semaphore.)

Again, all variations, such as `acceptor`, `acceptable`, and `accept_yo`, are allowed.

4.3. TEMPORAL CLAIMS [skip on a first reading]

We now come to the last, the most powerful, and the most rarely used, method for expressing correctness requirements on PROMELA validation models. Unavoidably, the complexity of the validations is higher for temporal claims than it is for any of the other requirements. Assertions and end-state labels can be checked most efficiently, then progress-state labels, followed by accept-state labels, and finally temporal claims. We will illustrate below, for instance, how temporal claims can be used to express linear-time temporal logic formulae.

The Never Primitive

The syntax of a temporal claim is as follows.

```
never { ...body... }
```

where `never` is a keyword, comparable to the keyword `proctype`, and `body` is a behavior specification. The `never` claim expresses behavior that is claimed to be *impossible*. A correctness violation occurs if and only if a temporal claim can be completely *matched* by a system behavior. There can be only one `never` claim per validation model.

Confusing at first may be that temporal claims do not specify independent system behavior, but they formalize claims about *existing* system behavior. The system behavior does not change when a temporal claim is added or removed: it is completely specified by the `init` clause and the `proctype` definitions.

Formally, a `never` claim defines a labeling of reachable system states with boolean propositions. Every statement in a temporal claim is therefore interpreted as a proposition, where the executability of the statement defines the truth-value.

Labeling System States

The mechanics of temporal claim matching can be explained as follows. The first statement (i.e., proposition) in the body of a temporal claim labels the first reachable system state of the validation model. (The system state that is reached *after* first statement in the `init` process has been executed.) For every new reachable system state along an execution path, the executability of the corresponding proposition of the temporal claim must be evaluated. Two things can happen.

- ☐ If the statement is executable, and thus the implicit proposition is true, the state of the temporal claim is updated with a move to its next statement (i.e., it ‘‘executes’’ the statement).
- ☐ If the statement is *not* executable, and thus the implicit proposition is false, this means that the behavior performed by the PROMELA validation model can *not* be matched by the claim. An automated validator, such as SPIN, will in this case truncate the search for errors (since no future pattern of system behaviors can lead to a complete match of the temporal claim) and continue the search by inspecting other reachable system states.

The temporal claim is completely matched if and when it reaches its normal end-state (the closing curly brace), meaning that all propositions held for the reachable system states along some execution path. The correctness requirement expressed by the temporal claim is then violated and an error has been found. (Note: it is not sufficient to reach an end-state label to match a claim. The match can, however, always be forced with a jump to the normal end state.)

Temporal claims can thus be used to trap illegal terminating behaviors. Below, we will also show how they can be used to catch illegal cyclic behaviors.

Two things make temporal claims a little harder to define than process behaviors.

- ☐ Every ‘‘statement’’ in a temporal claim must model a proposition, and therefore it must be side-effect free (i.e., no assignments, receive or send operations, etc.), that is, it is a pure condition. The propositions in a temporal claim do not (should not) define but ‘‘monitor’’ system behavior.
- ☐ To violate a claim, the series of propositions listed in a temporal claim must match the system behavior at every single step of system execution.

If, for instance, we want to say that whenever some proposition `P` becomes true, it is always the case that eventually another proposition `Q` becomes true, we would be tempted to express this claim as follows:

```
never {
        P -> Q
}
```

The syntax for the body of the claim is precisely that of `proctype` bodies (convenient, but potentially confusing, because the semantics of execution are different), with the exception that the propositions must be side-effect free. The above claim, though syntactically correct, does not express the right correctness requirement. It expresses that it would be an error if proposition `P` is true in the first reachable system state and `Q` is true in the state that *immediately* (not eventually) follows it.

Matching Behavior

If we want to allow for preceding or intermediate events, for instance for the possibility that `P` remains true for some amount of time, or toggles between true and false after it has become true at least once, we have to say so explicitly. This leads to the following revision of the claim

```
never {
S0:     do
        :: P || !P      /* or, equivalently: skip       */
        :: P -> break
        od;
S1:     do
        :: !Q
        :: Q -> break
        od;
S2:     skip            /* if reached: claim is matched */
}
```

This claim has three ‘‘states.’’ In the first state proposition `P` may be either true or false for an arbitrary number of steps. At any time when `P` is true, a transition can be made to the second state, which can only be left when `Q` is true. The temporal claim reaches its end-state only when both events happen.

Inverting a Claim

Temporal claims are named `never`, to indicate that they express behavior that should **never** happen. The temporal claim is **violated**, not satisfied, when they are matched (i.e., when the end state is reached). Our example correctness requirement stated that whenever some proposition `P` becomes true, it is always the case that eventually another proposition `Q` becomes true. The last claim we produced above, therefore, is still not quite right. (Not unlikely to happen in practice.) To get the right claim we must invert the above specification. To do this, notice that our correctness claim is only violated when the truth of `P` is *not* followed by the eventual truth of `Q`. This happens when Q remains false forever, in an infinite cycle, or the execution sequence terminates without Q ever becoming true. We already know how to match terminating behaviors, but not cyclic behaviors.

Matching Cyclic Behaviors

A cyclic behavior is matched in a temporal claim precisely as it is matched elsewhere in a PROMELA model: with acceptance labels. For example:

```
never {
        do
        :: skip
        :: P -> break
        od;
accept: do
        :: !Q
        od
}
```

Would trap a correctness violation when a cycle is detected through the accept label, as before. In fact, acceptance labels, progress state labels, and assertions can freely be used in temporal claims to trap errors. End-state labels, however, have no meaning inside temporal claims.

Adding a check for the second case, for terminating executions, produces the following extension:

```
never {
        do
        :: skip
        :: P -> break
        od;
accept: do
        :: !Q
        :: timeout && !Q -> break
        od
}
```

The keyword `timeout` has a special meaning in PROMELA. It becomes executable only when no further action is possible in a validation model. This is true in end-states and it can be used to model recovery from potential deadlock states (e.g., after message loss). In the above example, the `timeout` option only becomes executable (true as a proposition) when the system has terminated. Clearly, if it terminates while

`Q` is false, we have another correctness violation.

With some effort, temporal claims used in combination with acceptance-state labels can express also the absence of non-progress cycles. The claims are therefore more general than progress-state labels. The expense (complexity) of finding non-progress cycles directly with progress-state labels, however, is smaller than the expense of the validation of a claim that specifies the same property.

Linear-Time Temporal Logic

Zohar Manna and Amir Pnueli recently defined three classes of temporal logic formulae that they believed together cover ‘‘the majority of properties one would ever wish to verify.’’ [Manna & Pnueli ’90] The properties are named Invariance, Response, and Precedence. The quotes below are taken from [Manna & Pnueli ’90]. We give an example of the representation in PROMELA for each type of claim.

Invariance Claims

> ‘‘An invariance property refers to an assertion p, and requires that p is an invariant over all the computations of a program P, i.e., all the states arising in a computation of P satisfy p. In temporal logic notation, such properties are expressed by □ p, for a state formula p.’’

The corresponding temporal claim in PROMELA is as follows.

```
never {
        do
        :: p
        :: !p -> break
        od
}
```

where p is the required proposition.

Response Claims

> ‘‘A response property refers to two assertions p and q, and requires that every p-state (a state satisfying p) arising in a computation is eventually followed by a q-state. In temporal logic notation this is written as p => <> q.’’

The symbol => used here is not the logical implication symbol -> but the *entails* operator, defined as (p => q) == □ (p -> q). Thus the formula p => <> q means □ (p -> <> q).

The corresponding temporal claim in PROMELA is:

```
never {
        do
        :: skip
        :: p && !q -> break
        od;
accept:
        do
        :: !q
        od
}
```

Note that replacing the `skip` with `(!p)` would cause the claim to only capture violations of the response claim that follow the very first time proposition p becomes false (it may well toggle between true and false a few times before the violation occurs).

Precedence Claims

> ‘‘A simple precedence property refers to three assertions p, q, and r. It requires that any p-state initiates a q-interval (i.e., an interval all of whose states satisfy q) which, either runs to the end of the computation, or is terminated by an r-state. Such a property is useful to

> express the restriction that, following a certain condition, one future event will always be preceded by another future event. For example, it may express the property that, from the time a certain input has arrived, there will be an output before the next input. Note that this does not guarantee [require] that the output will actually be produced. It only guarantees [requires] that the next input (if any) will be preceded by an output. In temporal logic, this property is expressed by p => (q U r), using the unless operator (weak until) U.''

The corresponding temporal claim is:

```
never {
        do
        :: skip
        :: p && (!q && !r) -> goto error
        :: p && ( q && !r) -> break
        od;
        do
        :: q
        :: !q && !r -> goto error
        od;
error:  skip
}
```

Note again that the `skip` cannot be replaced with `(!p)` without altering the semantics of the claim. All types of linear-time temporal logic formulae can be expressed similarly in PROMELA. The most reliable option to perform this task is probably the preprocessor written by Brad Glade from Cornell University [Glade '91]. Glade's program translates arbitrary temporal logic formulae directly into PROMELA `never` claims.

Remote Referencing

To get the full benefit of temporal claims, we must be able to refer to the control-flow states and the variable values of running processes. As an example, consider the following claim, referring to a protocol system with at least one message channel called `receiver`, a message type called `msg0`, and a process type called `Receiver` containing at least two statements, labeled `P0` and `P1` respectively.

```
never {
        do
        :: len(receiver) == 0
        :: receiver?[msg0] -> goto accept0
        :: receiver?[msg1] -> goto accept1
        od;
accept0:
        do
        :: !Receiver[2]:P0
        od;
accept1:
        do
        :: !Receiver[2]:P1
        od
}
```

This claim corresponds to a machine with 4 states: the initial state, the two states that were labeled, and the normal end state. At least one of three conditions must be true in the initial system state. The claim remains in this state as long as channel `receiver` is empty. If it contains a message `msg0` or `msg1` it will change state to either `accept0` or `accept1`, depending on the message that was matched. Once the transition to, for instance, state `accept0` has been made, the claim can only remain in that state if the receiver process will never accept a message with the same sequence number, i.e., if the receiver process never passes the state labeled `P0`.

There can be many instantiations of the process type `Receiver` so we need some way of specifying exactly which particular instantiation we mean when we refer to the state of a process. This is the only time that we need to be able to refer to the *instantiation number* or the `pid` of a process. The `pid` of a process is the number that is returned by the `run` operator, when a process is instantiated. The `pid`s are assigned in the order in which processes are started, but they may be recycled when processes die. The initial process always has pid zero, and its number is never recycled. A `pid` can usually easily be inferred from the program text. Since the receiver process is the second process that is instantiated in this system, its `pid` is two. We can refer to the receiver process unambiguously as `Receiver[2]`. The condition that the receiver is currently in the state labeled `P0` is expressed as `Receiver[2]:P0`. The condition is false whenever the second process that was instantiated is in any state other than at label `P0` of process type `Receiver`.

The notation `Receiver[2]:P0` is a special case of a more general construct which allows temporal claims to refer to internal conditions of the asynchronous processes defined within a PROMELA model. A reference to the current value of local variable `any` in the receiver process, for instance, is written `Receiver[2].any`. It can be used in arbitrary expressions, such as

```
assert(Receiver[2].any < 0)
```

In process references, a colon is used to refer to labels (i.e., control flow states) and a period is used to refer to local variables. In the first case, the value returned is a Boolean. In the second case it is the integer value of the variable specified.

5. AN EXAMPLE VALIDATION

Lynch’s Protocol

With these new tools, we can now validate the correctness of our example protocol more thoroughly. We have already shown that no message is accepted more than once by B, using an `assert` statement. This satisfies the second half of the correctness claim that was made on lines 32-33 of the informal specification. To prove also the first half, we must show that:

- ○ Every message fetched by A is received at least once by B.

The execution sequences we are looking at here are cyclic, that means that we can use progress labels, acceptance labels, or temporal claims. Non-progress loops are usually identified by a process of elimination. By default (i.e., in the absence of progress labels) all execution cycles are considered to be non-progress loops, and reported as correctness violations. We can eliminate cycles from consideration by identifying progress statements in the process bodies. For instance, the statements corresponding to the acceptance of a correct message in B are clearly progress statements. We can mark any one of them to rule out cycles in which messages are accepted. This leads to the following revised specification for B.

```
proctype B(chan in, out)
{    byte mr;                        /* message data              */
     byte last_mr;                   /* mr of last error-free msg */
     bit  ar;                        /* alternation bit           */
     bit lar=1;                      /* ar of last error-free msg */
     bit ini;                        /* lines 27-31   */

     do
     :: in?error(mr,ar) ->           /* lines 7-10    */
             out!control(0)          /* lines 8,25,26 */
     :: in?data(mr,ar) ->            /* lines 7-10    */
             out!control(1);         /* lines 8,25,26 */
             if                      /* lines 27-31   */
             :: (!ini) ->            /* lines 27-31   */
                     ini = 1;        /* lines 27-31   */
                     lar = ar        /* lines 27-31   */
             :: ini ->               /* lines 27-31   */
                     if                /* line 20     */
                     :: (ar == lar) -> /* line 21     */
                             skip      /* line 23     */
                     :: (ar != lar) -> /* line 24     */
progress:                    ACCEPT;   /* line 24     */
                             lar = ar; /* line 22     */
                             last_mr = mr
                     fi
             fi                      /* lines 27-31   */
```

```
    od
}
```

If we do the validation for non-progress cycles, however, we immediately discover that the correctness claim can be violated when messages can be distorted infinitely often. There is a cycle of events where A send a message, C distorts it, and B rejects it as an error, A retransmits the message, etc.

After recognizing that this particular scenario is also not of interest to us in the search for violations of the correctness criterion, we can eliminate it from the search as well by labeling another statement. The simplest way is to label the statements in the channel process C.

```
 proctype lower(chan fromA, toA, fromB, toB)
 {   byte d; bit b;

     do
     :: fromA?data(d,b) ->
             if
             :: toB!data(d,b)        /* correct   */
             ::
progress0:      toB!error            /* distorted */
             fi
     :: fromB?control(b) ->
             if
             :: toA!control(b)
             ::
progress1:      toA!error
             fi
     od
 }
```

Of course, this labeling should not be interpreted to state that message distortion is desirable. It merely formalizes the designer’s intention to ignore in the validation any cycle that involves an infinite repetition of message distortion (the probability of such a sequence would be infinitely small...). Note also that labels are placed after the double-colon flags, and not before them (a label must always prefix a statement).

With these extensions the validation for non-progress cycles proceeds as follows. It is also wise to leave the `printf` statements in `ACCEPT` statements disabled.

```
$ spin -a lynch01
$ cc -o pan pan.c
```

The executable analyzer is compiled as before. The search for non-progress loops is listed among the available options of the executable validator. Check it as follows.

```
$ pan -?        # check the options
unknown option
-cN stop at Nth error (default=1)
-l  find non-progress loops
-mN max depth N (default=10k)
-wN hashtable of 2ˆN entries (default=18)
$ pan -l
full statespace search for:
        assertion violations and non-progress loops
vector 88 byte, depth reached 399, non-progress loops: 0
    4433 states, stored
       4 states, linked
    5039 states, matched           total:     9476
hash conflicts: 2107 (resolved)
(max size 2ˆ18 states, stackframes: 0/108)
$
```

Good news. In a full state space search (with 100% coverage) no non-progress loops were found, which proves the correctness requirement we stated. To illustrate also the usage of temporal claims, suppose we wanted to show that in the absence of message distortion, there can be no infinite stream of duplicate messages, without any correct message getting through. The proof is implied by the last validation run, but we can try to express it in a different way with a temporal claim.

First, we mark line 20 in the specification for B above with the label `Dup` (the name is irrelevant).

```
 1  never {
 2      /* there is no cycle through label ‘‘Dup’’ in B that
 3       * doesn’t also pass through label ‘‘progress’’
 4       */
 5  accept:     do
 6              ::      do
 7                      ::!B[2]:Dup && !B[2]:progress
 8                      :: B[2]:Dup -> break
 9                      od;
10                      do
11                      :: B[2]:Dup
12                      ::!B[2]:Dup && !B[2]:progress -> break
13                      od
14              od
15  }
```

We have to be careful again to specify a complete set of propositions, covering all intermediate states. The specification above claims that all cycles are invalid that consist of a prefix of an arbitrary number of states in which process B is at neither of the two labels (lines 6-9). The transition to the second half of the claim can only happen when process B is at the label ‘‘Dup’’ (line 8), to make sure that the cycle contains at least one such occurrence. The cycle is completed when B leaves the ‘‘Dup’’ state again without passing through ‘‘progress’’ (line 12).

The validation run shows that this scenario is not feasible. No error is reported, confirming that the scenario is indeed impossible:

```
$ spin -a lynch02
$ cc -o pan pan.c
$ pan -?        # presence of accept labels disables loop analysis option:
unknown option
-cN stop at Nth error (default=1)
-mN max depth N (default=10k)
-wN hashtable of 2^N entries (default=18)
$ pan
full statespace search on behavior restricted to claim for:
        assertion violations
        and absence of acceptance labels in all cycles
vector 88 byte, depth reached 81, errors: 0
     263 states, stored
       2 states, linked
     196 states, matched           total:      461
hash conflicts: 76 (resolved)
(max size 2^18 states, stackframes: 0/13)
```

6. LARGE VALIDATION PROBLEMS

Protocol validation models of practical significance can quickly reach a size that is well beyond what can be analyzed with a traditional reachability analysis method (i.e., the default analysis method of SPIN discussed so far). Consider, for instance, a standard flow control protocol for selective retransmission. A simple validation model for this protocol derived in [Holzmann ’91] includes three processes (sender, receiver and channel process) and four message queues of two slots each. Each system state takes 160 bytes of memory to store. The table below lists the total number of reachable states for three different types of assumptions about the transmission channel, and the resulting size of a complete state space in Megabytes.

Assume we want to perform the three validations on a large machine with 128 Megabyte of storage. A traditional (exhaustive) reachability analysis can give complete coverage of the state spaces for the first two cases. For the third case the maximum coverage that can be obtained seems to be limited to the fraction 128/990.1 or 12.92%.

Alas, maintaining the state space on disk instead of in main memory is no real solution. A disk-based algorithm is three to four orders of magnitude slower than an in-memory algorithm [Holzmann ’91]. In practice

Table 6.1 — State Space Sizes for a Sample Problem

Assumptions about Channel	Number of Reachable States	State Space Size
Ideal Channel	90,845	14.5 Mbyte
Message Loss	396,123	63.4 Mbyte
Loss + Duplications	6,188,322	990.1 Mbyte

this means that even on the fastest machines a disk-based algorithm does not run faster than about 10 states per second, giving a runtime of more than 7 days to generate six million states.

Compiling the analyzers generated by SPIN with a single extra flag produces an executable that is optimized for large state spaces (see also the Exercises).

```
$ cc -DBITSTATE -o pan pan.c    # compile a supertrace analyzer
$ pan                           # run it as before
...
```

A supertrace search hashes the six million states into the one billion available bits and achieves a measured coverage of 99.99% for the same problem, taking no more than a few minutes of CPU time.

It can easily be measured that a supertrace search gives a superior coverage of the state space for all protocol problems where the total state space size is larger than available memory, independent of the protocol or the machines used. The hashing mechanism that is used provides a random sampling of states from the reachable state space, with a high probability of catching errors, should they exist. For details of the supertrace algorithm and its implementation in SPIN, we refer to [Holzmann ’91].

With the exception of acceptance validation labels, all correctness requirements can be checked with a supertrace search (including, for instance, a fast search for non-progress cycles). Clearly, the default search mode of SPIN, a traditional reachability analysis, is the preferred method when the estimated size of the reachable state space is less than the amount of available memory. In all other cases, a supertrace search will be superior.

7. EXERCISES

7.1. Evaluating Search Complexity

This first problem is meant to give an indication (appreciation?) of the state space explosion phenomenon, and how SPIN deals with it. It consists of eight small exercises using SPIN, numbered **[1.a]** through **[1.h]**. At each step, try to predict what you think should happen, write down your predictions, perform the experiment, and explain what happened. Do not continue until you understand precisely what you observed.

[1.a] How many reachable states do you predict will the following naive PROMELA system generate?

```
init {  /* file: ex.1a */
        byte i = 0;
        do
        :: i = i+1
        od
}
```

Try a simulation run:

```
$ spin -p -l ex.1a      # print out local vars at every step
...
```

Will the simulation terminate? □

[1.b] Estimate the total number of reachable states that should be inspected in an exhaustive validation. Is it a finite number? Will a validation run terminate? Try it as follows.

```
$ spin -a ex.1a
$ cc -o pan pan.c
$ pan
```

```
...
```

Explain the output in detail. □

[1.c] What would happen if you had declared the variable to be a `short` instead of a `byte`? What if you use an `int`? [Hint: use Table A.2 from Appendix A.]

Try either of them with a validation run. How do you explain the result? [Hint: check the analyzer's defaults by saying `pan -?`] □

[1.d] Next, predict accurately how many reachable states there are for this system. Write them down as a complete reachability tree.

```
#define N       2
init {  /* file: ex.1b */
        chan dummy = [N] of { byte };
        do
        :: dummy!85
        :: dummy!170
        od
}
```

Check your prediction as follows. We're only interested in the size of the state space, not in the obvious problems caused by buffer overflow, so we use the `-m` option from SPIN to define that buffer overflow is to be ignored. (Messages appended to a full buffer are then lost.)

```
$ spin -m -a ex.1b      # use -m to ignore buffer overflow
$ cc -o pan pan.c
$ pan
...
```

Explain the result. □

[1.e] What happens if you set N to 3 ? Express the number of states as a function of N. Use the formula to calculate how many states there will be if you set N to 14 ? Check your prediction as follows.

```
$ spin -m -a ex.1b      # use -m to ignore buffer overflow
$ cc -o pan pan.c       # optional: use the optimizer -O
$ time pan
...
```

Write down:

T: the *sum* of user time plus system time for the run
S: the number of states *stored*
G: the number of *total* number of states generated and analyzed
V: the vector-size (the amount of memory needed to store one state)

G/T gives you an accurate measure for the efficiency of the run. (for a kick: do the same experiment on other validation systems, if you have access to them).

S*V gives you the amount of memory that was used to store the state space. This is not the only place where memory that is used during the search (the stack also consumes memory) but it is typically the largest memory requirement. □

[1.f] The efficiency of the conventional reachability analysis is determined by the state space storage functions. To study this, repeat the last validation run with a smaller and a bigger hash table for the state space:

```
$ time pan -w10         # hash table with 2ˆ10 slots
...
$ time pan -w20         # hash table with 2ˆ20 slots
...
```

Explain the results. [Hint: compare the number of hash conflicts.] □

[1.g] How much memory would you need to do a run with N=20 ? (Warning: both the number of reachable states and the number of bytes per state goes up with N. Estimate about 30 bytes per state for N=20.)

Given that you have about 8 Megabyte on your system, what maximal fraction of the state-space would you expect to be able to analyze ?

Now set N to 20 and perform a supertrace validation, as follows.

```
$ spin -m -a ex.1b              # as before
$ cc -DBITSTATE -o pan pan.c    # different
$ time pan
...
```

If you did the calculation, you probably estimated that there should be 2,097,151 reachable system states for N=20. What percentage of these states was reached in the supertrace run? How much memory was used [Hint: cf. **[1.h]** below] ? (Compare to the earlier estimated maximal coverage for a conventional validation and explain the difference.) □

[1.h] The default bit-state space in a supertrace, used above, run has 2ˆ22 bits (i.e., 2ˆ18 bytes, or about one quarter Megabyte) repeat the run with different amount of memory to get different coverage. Check what percentage of the number of states is reached when you use the 8 Megabyte state space on which your first estimate for maximal coverage in a full state space search was based (2ˆ23 bytes is 2ˆ26 bits, which means a runtime flag `-w26`)

7.2. Validation of a Protocol

This problem let's you apply the validation strategies from the paper to a protocol that is very similar to the informal example that was discussed. Figure 3 shows a protocol specification given by Bartlett et al. in their seminal paper introducing the alternating-bit protocol [Bartlett et al. 1969, Figure 3c]. Use SPIN to see if it meets the same correctness criteria as the informal protocol defined by Lynch, discussed earlier in this paper. □

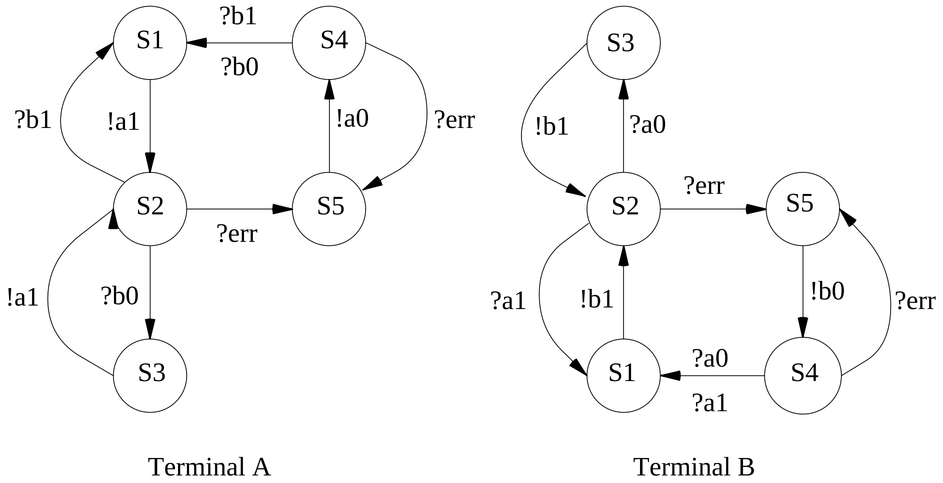

Figure 3 — A Half-Duplex File Transfer Protocol from [Bartlett et al. '69]

7.3. Validation of an Interface Standard

It is time to validate an international standard. CCITT Recommendation X.21 has the dubious honor of being one of the first protocols that was shown to be incompletely specified with an automated analysis. The validation was performed in 1977 by Colin West and Pitro Zafiropulo [West et al. 1978].

What in 1977 was described as a ''reasonably complex protocol'' has become, after a development of almost 15 years, a rather trivial litmus test for automated protocol validators, that shouldn't take more than a few milli-seconds of CPU time. Validate the X.21 protocol using SPIN. Derive the validation model from Figure 4, which is based on the original specification used in [West et al. 1978]. (''*all*'' in Figure 4 means ''all other states.'') □

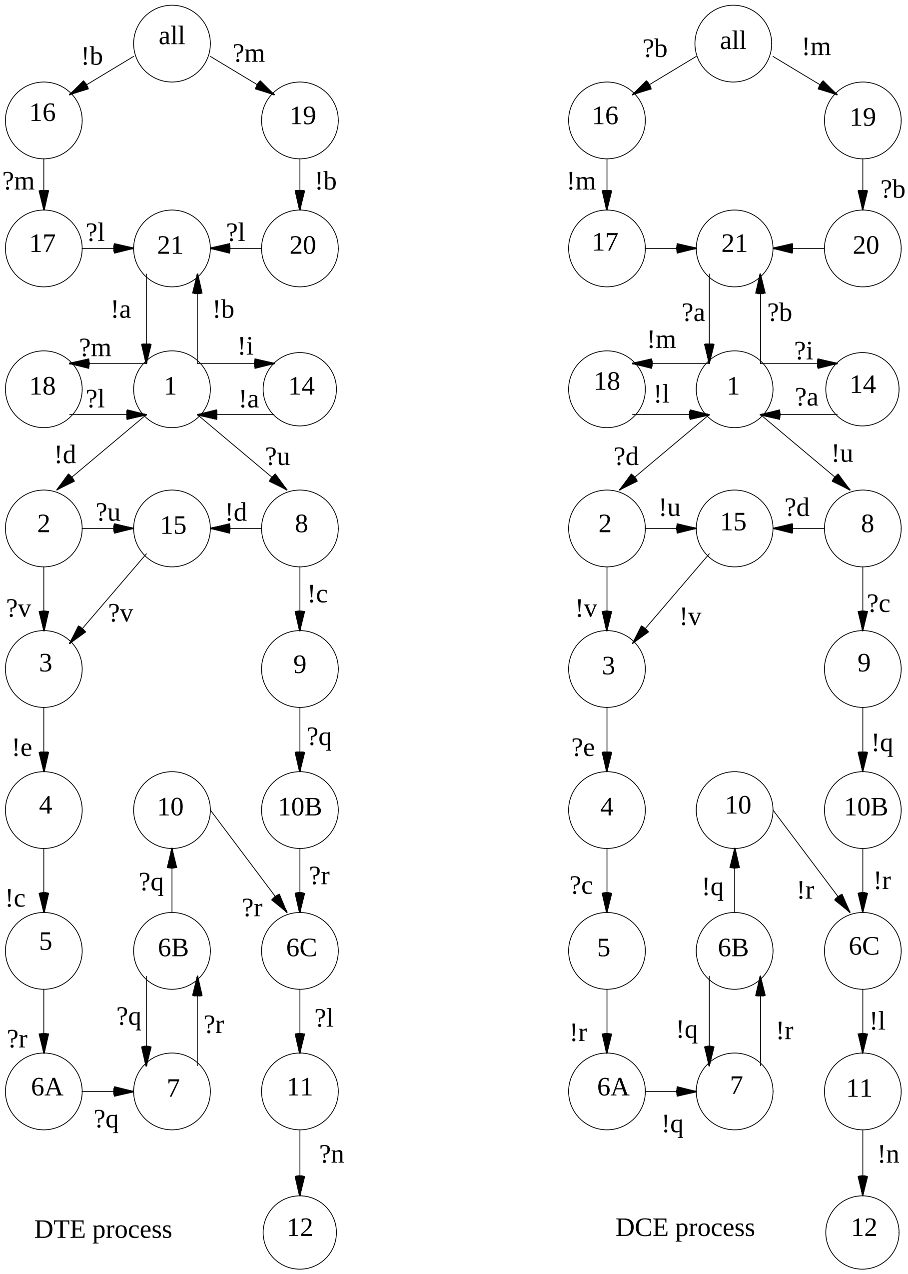

Figure 4 — X.21 Specification (1977)

7.4. Validation of Mutual Exclusion Algorithms

If two or more concurrent processes execute the same code and access the same data, there is a potential problem that they may overwrite each others results and corrupt the data. The mutual exclusion problem is the problem of restricting access to a critical section in the code to a single process at a time, assuming *only* the indivisibility of read and write instructions. (The problem becomes trivial if you can assume an indivisible test-and-set instruction.) The problem was first posed by Dijkstra in [Dijkstra 1965].

[4.a] The following ‘improved’ solution appeared one year later in the same journal (*Comm. of the ACM*, Vol. 9, No. 1, p. 45) by another author. It is reproduced here as it was published (in pseudo Algol).

```
Boolean array b(0;1) integer k, i, j,
comment process i, with i either 0 or 1;
```

```
C0: b(i) := false;
C1: if k != i then begin
C2: if not (b(j)) then go to C2;
    else k := i; go to C1 end;
    else critical section;
    b(i) := true;
    remainder of program;
    go to C0;
    end
```

Modeling the solution in PROMELA, and proof or disproof the correctness of the algorithm. How long did it take you? □

[4.b] The problem continues to be popular. For an overview solution attempts up to roughly 1984, see Raynal [1984/1986] or [Lamport 1986]. For two processes, for instance, a particularly elegant solution was published by G.L. Peterson [Peterson 1981]. In PROMELA the solution can be modeled as follows.

```
#define true    1
#define false   0
bool flag[2];
bool turn;
proctype user(bool i)
{       flag[i] = true;
        turn = i;
        (flag[1-i] == false || turn == 1-i);
crit:   skip;   /* critical section */
        flag[i] = false
}
init { atomic { run user(0); run user(1) } }
```

Prove Peterson's algorithm correct with SPIN. □

[4.c] Alas, despite all the publications, even today there is no shortage of faulty solutions to the mutual exclusion problem. The next version was recommended only recently by a major computer manufacturer in the U.S. to a client. (Name omitted to protect the guilty.)

```
byte in;
byte x, y, z;
proctype user(byte me)
{
L1:     x = me;
L2:     if
        :: (y != 0 && y != me) -> goto L1      /* try again */
        :: (y == 0 || y == me)
        fi;
L3:     z = me;
L4:     if
        :: (x != me)  -> goto L1                /* try again */
        :: (x == me)
        fi;
L5:     y = me;
L6:     if
        :: (z != me) -> goto L1                 /* try again */
        :: (z == me)
        fi;
L7:                                             /* success */
        in = in+1;
        assert(in == 1);
        in = in - 1;
        goto L1
}
init { atomic { run user(1); run user(2) } }
```

First convince yourself that the algorithm is correct. Next show that it is faulty with SPIN. How long did each step take you? □

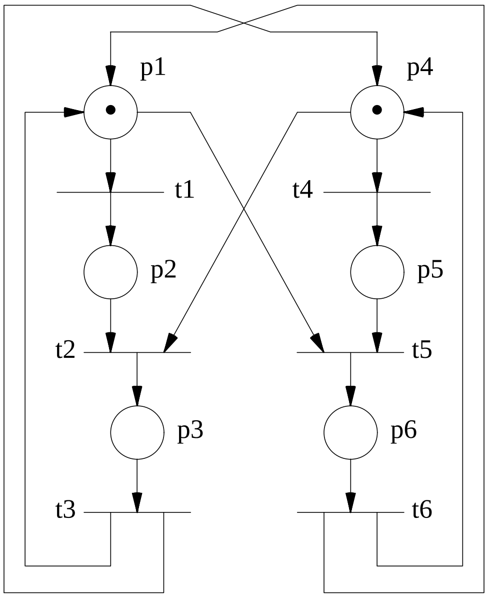

Figure 5 — Petri Net with Hang State

7.5. Validation of Petri Nets

It is often much easier to build an little validation model and mechanically verify it than it is to understand a manual proof of correctness in detail. Petri Nets are relatively easy to model as PROMELA validation models. A PROMELA model for the net in Figure 5, for instance, is quickly made.

```
#define Place   byte    /* assume < 256 tokens per place */

Place p1, p2, p3;
Place p4, p5, p6;
#define inp1(x)         (x>0) -> x=x-1
#define inp2(x,y)       (x>0&&y>0) -> x = x-1; y=y-1
#define out1(x)         x=x+1
#define out2(x,y)       x=x+1; y=y+1
init
{       p1 = 1; p4 = 1; /* initial marking */
        do
/*t1*/  :: atomic { inp1(p1)    -> out1(p2) }
/*t2*/  :: atomic { inp2(p2,p4) -> out1(p3) }
/*t3*/  :: atomic { inp1(p3)    -> out2(p1,p4) }
/*t4*/  :: atomic { inp1(p4)    -> out1(p5) }
/*t5*/  :: atomic { inp2(p1,p5) -> out1(p6) }
/*t6*/  :: atomic { inp1(p6)    -> out2(p4,p1) }
        od
}
```

For this exercise, consider the Petri Net published as Figure 1 in [Berthelot and Terrat 1982]. (Copies available at the tutorial session.) The net was proven to be deadlock free with a manual reduction and proof technique in the paper. Build a model in PROMELA and find the deadlock with SPIN. □

8. SUMMARY

There are several ways of expressing correctness requirements in PROMELA, ranging from simple, but frequently used, requirements for specifying logical soundness and completeness criteria (absence of deadlocks, unspecified receptions), to more subtle liveness conditions (absence of non-progress loops, absence of livelocks), to temporal claims. The simpler claims are easy to express and can be validated efficiently. As can be expected, the more complicated claims require more thought and increase the run-time requirements for an automated validator.

In the initial stages of a design, a user is unlikely to use more than assertions and perhaps end-state labels. In the final stages of a design, when all initial flaws have been corrected and a more precise qualitative assessment of the design can be made, validations with explicit temporal claims may be developed. In many cases this level of sophistication in a validation is never required and all the necessary properties can be established without it.

It is almost impossible to manually verify correctness requirements such as the ones we have discussed, no matter how diligent or disciplined the designer. The behavior of even simple protocol systems can be of a complexity that no designer can be expected to assess accurately.

There are several ways to approach this problem. The following quote illustrates how Korean typesetters solved it in the 15th Century.

> ''The supervisor and compositor shall be flogged thirty times for an error per chapter; the printer shall be flogged thirty times for bad impression, either too dark or too light, of one character per chapter.''
>
> *Early Movable Type in Korea* -- Kim Won-Yong, 1954

The quote appears in a book by Daniel Boorstin. He observes:

> ''This helps explain both the reputation for accuracy earned by the earliest Korean imprints and the difficulty that Koreans found in recruiting printers.''
>
> *The Discoverers* -- Chapter 62, D.J. Boorstin, 1983

In protocol design at least, a better approach seems to be to provide designers the tools that can help them secure the desired accuracy. Such tools are needed not only to *express* the correctness requirements of a protocol design, but also to verify those requirements reliably. The first efficient tools for protocol validation are becoming available. SPIN is such a tool, and it is likely to be followed by many others supporting different types of languages and systems.

One of the main strengths of SPIN is that it can work on any hardware. It explicitly takes the constraints of a machine into account, be it a Cray, a VAX, or a PC, and delivers the best results possible. SPIN is available for educational purposes at no charge.

9. REFERENCES

Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. [1969], ‘‘A note on reliable full-duplex transmission over half-duplex lines,’’ *Comm. of the ACM*, Vol. 12, No. 5, 260-265.

Berthelot, G., and Terrat, R. [1982], ‘‘Petri Net Theory for the correctness of protocols,’’ IEEE Trans. on Comm., Vol COM-30, No. 12, Dec. 1982, pp. 2497-2505.

Dijkstra, E.W. [1965], ‘‘Solution to a problem in concurrent programming control,’’ *Comm. of the ACM*, Vol 8, No. 9, p. 569.

Dijkstra, E.W. [1975], ‘‘Guarded commands, nondeterminacy and formal derivation of programs,’’ *Comm. of the ACM*, Vol. 18, No. 8, pp. 453-457.

Glade, B.B. [1991], ‘‘A temporal logic to Promela ‘‘never clause’’ converter,’’ EE594 final project, Cornell University, Electrical Engineering Dept., May 3, 1991.

Hoare, C.A.R. [1978], ‘‘Communicating sequential processes,’’ *Comm. of the ACM*, Vol. 21, No. 8, pp. 666-677.

Holzmann, G.J. [1991], *Design and validation of computer protocols*, Prentice Hall, 512 pgs, ISBN 0-13-539925-4, International Edition, ISBN 0-13-539834-7. [Refer to the above for more complete bibliographic notes on the subject of this tutorial and access to the software.]

Lamport, L. [1986], ‘‘The mutual exclusion problem — parts I and II", *Journal of the ACM*, Vol. 33, No. 2, April 1986, pp. 313-347.

Lynch, W.C. [1968], ‘‘Reliable full duplex file transmission over half-duplex telephone lines,’’ *Comm. of the ACM*, Vol. 11, No. 6, pp. 407-410.

Manna, Z., and Pnueli, A. [1990], *Tools and Rules for the Practicing Verifier*, Stanford University, Report STAN-CS-90-1321, July 1990, 34 pgs.

Peterson, G.L. [1981], ‘‘Myths about the mutual exclusion problem,’’ Letters,’’ ‘‘*Inf.* Proc. Vol. 12, No. 3, pp. 115-116.

Raynal, M. [1984/1986], *Algorithms for Mutual Exclusion*, MIT Press, Cambridge, Mass., ISBN 0-262-18119-3, 107 pgs. [The 1986 edition is a translation from the original French version published in 1984.]

Schwartz, L.S. [1963], ‘‘Feedback for error control and two-way communication,’’ *IEEE Trans. CT*, March 1963, pp. 49-56.

APPENDIX A — BRIEF REFERENCE MANUAL FOR PROMELA

This appendix gives an overview of the main syntax requirements of the language. Semantics and usage are more fully explained in the paper. This manual does not cover possible restrictions or extensions of specific implementations of the validator SPIN. It has the same machine dependencies as the programming language C. In case of doubt, for instance, when you have to find out what the precise effect is of an expression such as `(-10)%(-9)` or `(-10)<<(-2)` on your machine, the quickest way to learn is to execute a little PROMELA test program, like

```
init { printf("%d\t%d\n", (-10)%(-9), (-10)<<(-2)) }
```

The meaning of all conventional operators matches that of ANSI-C.

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 cannot be nested.

IDENTIFIERS

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

KEYWORDS

The following identifiers are reserved for use as keywords:

```
assert          atomic          bit             bool
break           byte            chan            do
fi              goto            if              init
int             len             mtype           never
od              of              printf          proctype
run             short           skip            timeout
```

CONSTANTS

There are three types of constants.

- String constants
- Enumeration constants
- Integer constants

String constants can only be used in `printf` statements.

Enumeration constants can be used to define symbolic names for message types. They can be defined in `mtype` declarations of the type

```
mtype = { namelist }
```

where `namelist` is a comma separated list of symbolic names. Only one `mtype` declaration per program can be used.

An integer constant is a sequence of digits representing a decimal integer. There are no floating point numbers in PROMELA.

EXPRESSIONS

The evaluation of expressions is defined in integer arithmetic. Unsigned data, that is all variables declared with type `bit`, `byte`, or `bool`, are cast to signed integers before being used in expressions. For example, the value of expression `(p-1)`, with `p` a variable of type byte (unsigned char) and value zero, is the signed value `-1` in PROMELA, and not the unsigned equivalent 255. On assignments, however, the type of the destination always prevails. The value `-1` is cast to 255 when it is stored in a unsigned variable, but it remains `-1` when stored in a signed variable.

The following operators can be used to build expressions.

`+, –, *, /, %,`	arithmetic operators
`>, >=, <, <=, ==, !=,`	relational operators
`&&, ||, !`	logical AND, OR, NOT
`&, |, ˜, >>, <<`	`C`-style bit operators
`!, ?`	send and receive operators
`(), []`	grouping, indexing
`len, run`	special operators

The syntax, semantics, side effects, and machine dependencies of all operators match ANSI standard C. Table A.1 defines the precedence levels. The operators on the first line in the table have the highest precedence.

Table A.1 — Precedence and Associativity

Operators	Associativity
`()` `[]`	left to right
`˜` `-` *(unary minus)* `!` *(boolean negation)*	left to right
`*` `/` `%`	left to right
`+` `-`	left to right
`>>` `<<`	left to right
`>` `<` `>=` `<=`	left to right
`==` `!=`	left to right
`&`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`!` *(send)* `?` *(receive)*	left to right
`len` `run`	left to right
`=`	right to left

Most operators, including assignment =, take two operands. The boolean negation ! and the unary minus – operator can be both unary and binary, depending on context. The assignment operator takes an expression on the right, and a variable reference on the left:

```
varref = expression
```

Unlike C, the assignment operator cannot be used in expressions in PROMELA. The unary operator, `len`, applies to message channels only, and the unary operator `run` applies to process types. Informally, we talk about `len` or `run` statements, and similarly about send and receive statements, for statements that contain these operators.

REMOTE REFERENCING

Global variables and local variables declared within the same process type can be referred to by name. For instance

```
byte glob;

proctype same()
{       bool loc;

here:   (loc+glob)
}
```

Local variables of other processes can be referred to as follows:

```
proctype other()
{
```

```
          assert(same[2].loc > 3)
   }
```

Here a process of type `other` refers to the local variable `loc` of the process with `pid` two, i.e., the second process that was instantiated. It is a run-time error if the type of that process is different from the specified type `same`.

The process state of a remote process can be tested with boolean colon expressions. For instance, the condition

```
   same[2]:here
```

is true if and only if the process referred to is currently in the state that was labeled `here`. Remote referencing of variables and control flow states is intended to be used only in assertions and in temporal claims. The language definition, however, does not prevent other applications.

DECLARATIONS

Processes and variables must be declared before they can be used. Variables can be declared either locally, within a process type, or globally. A process can only be declared globally in a `proctype` declaration. `Proctype` declarations cannot be nested. Local declarations may appear anywhere in a process body. The scope of a local variable is the complete process body, irrespective of where its declaration is placed. It is not accessible, though, until execution has passed the point of declaration at least once. Ther five basic data types are listed in Table A.2.

Table A.2 — Basic Data Types

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

VARIABLES

A variable declaration begins with a keyword indicating the data type of the variable followed by a list of identifier names, each one optionally followed by an initializer.

```
   byte name1, name2 = 4, name3;
   chan qname; chan a = [3] of { byte };
```

The initializer must be an expression for a variable of a basic data type, and a channel specification for variables of type `chan`. By default, variables of all types except `chan` are initialized to zero. Variables of type `chan` must be initialized explicitly before they can be used for message passing. It is undefined what the result is of using an uninitialized channel variable. Most likely, it causes a fatal runtime error.

Table A.2 summarizes the width and attributes of the five basic data types. 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];
   chan q[M];
```

where `N` and `M` are constants. An array declaration may have an initializer, which initializes all elements of the array. If the array is a channel, one message channel of the given type per array element is created. In the channel initializer

```
   chan q[M] = [x] of { types }
```

`M` is a constant, `x` is an expression that specifies the size of the channel, and `types` is a comma separated list of one or more data types that defines the format of each message that can be passed through the channel.

All channels are initialized to be empty. Initialized channel identifiers can be passed from one process to another in messages or in `run` statements.

PROCESSES AND TEMPORAL CLAIMS

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 a process declaration is enclosed in parentheses.

```
proctype name( /* parameter declarations */ )
{
        /* declarations and statements */
}
```

The parameter declarations cannot have initializers. One process declaration is required in every PROMELA model: the initial process. It is declared without the keyword `proctype` and without a parameter list.

```
init {  /* declarations and statements */       }
```

It is the first process running and it has `pid` zero.

A temporal claim starts with the keyword `never` and can contain any PROMELA text

```
never { /* declarations and statements */       }
```

There can be at most one temporal claim per PROMELA model. It is used to specify a correctness requirement about the executions of the system specified. The temporal claim specifies a behavior that is claimed to be impossible. The claim will normally only contain conditions, though it is valid to allow the temporal claim to contain variable declarations, atomic sequences, and send and receive statements. To violate a correctness claim, it must be possible to execute one statement, or one atomic sequence of statements, for every statement that is executed by any of the other processes in the model. By using the temporal claim in combination with acceptance-state labels, any linear-time propositional temporal logic formula on the system behavior can be expressed (see Chapter 6).

STATEMENTS

There are twelve types of statements:

```
assertion       assignment      atomic          break
expression      goto            printf          receive
selection       repetition      send            timeout
```

Any statement can be preceded by one or more declarations. 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 evaluation of a compound expression is always indivisible. This means that the statement

```
(a == b && a != b)
```

will always be unexecutable, but the sequence

```
(a == b); (a != b)
```

may be executable in that order.

The act of passing the statement after a successful evaluation is called the ‘‘execution’’ of the statement. There is one *pseudo* statement, `skip`, which is equivalent to the condition `(1)`. `Skip`, is a null statement; it is always executable and has no effect when executed. It may be needed to satisfy syntax requirements.

Goto statements can be used to transfer control to any labeled statement within the same process or procedure. They are always executable. Assignments and declarations are 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 is always executable.

Each statement may be preceded by a label: a name followed by a colon. Each label may be used as the destination of a `goto`. Three types of labels have predefined meanings in validations: end-state labels, progress-state labels, and acceptance-state labels. The semantics are explained in the paper.

The remaining statements, selection, repetition, send, receive, break, timeout, and atomic sequences, are discussed below.

SELECTION

A selection statement begins with the keyword `if`, is followed by a list of one or more options and ends with the keyword `fi`. Every option begins 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. Thus the language defines nondeterministic machines.

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 transfers control outside the cycle. The keywords of the repetition statement are `do` and `od` instead of `if` and `fi`. 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 SEQUENCE

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 in an atomic sequence other than the first one is found to be unexecutable. The first statement is called the *guard* of the sequence. If it is executable, so should be the rest of the sequence. In general, therefore, the guard of an atomic sequence is followed only with local assignments and local conditions, but not with any send or receive statements.

SEND

The syntax of a send statement is

```
q!expr
```

where `q` is the name of a channel, and the evaluation of expression `expr` returns a value to be appended to the channel. The send statement is not executable (blocks) if the 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:

```
q!expr1,expr2,expr3
```

Equivalently, this may be written

```
q!expr1(expr2,expr3)
```

RECEIVE

The syntax of the receive statement is

```
q?name
```

where `q` is 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,

```
q?name1,name2,...
```

which again is equivalent to

```
q?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. It is an error to attempt to receive a value when none was transferred, and vice versa.

Any receive statement can be used as a side-effect free condition by enclosing its parameter list in square braces:

```
q?[name1,name2,...]
q?[name1(name2,...)]
```

The statement is executable (returns a non-zero result) only if the corresponding receive operation is executable, but it has no effect on the variables or the channel.

The only other type of operation allowed on channels is

```
len(varref)
```

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

TIMEOUT

The keyword `timeout` represents a condition that becomes true if and only if no other statement in the system is executable. A timeout statement has no effect when executed. Timeouts can be included in expressions.

MACROS AND INCLUDE FILES

The source text of a specification is processed by the C preprocessor for macro-expansion and file inclusions, Kernighan and Ritchie [1978].

APPENDIX B — PROMELA GRAMMAR

The grammar is listed in BNF-style. Parenthesis are used for grouping. A plus indicates a repetition of one or more times of the last syntactical unit; a star indicates a repetition of zero or more times. Square brackets are used to indicate optional elements. A vertical bar separates options. Literals are quoted. Terminals are upper-case, non-terminals are lower-case.

```
program    ::= { unit } +

unit       ::=  PROCTYPE NAME ’(’ [ decl_lst ] ’)’ body
           | CLAIM body
           | INIT body
           | one_decl
           | MTYPE ASGN ’{’ NAME { ’,’ NAME } * ’}’

body       ::=  ’{’ sequence ’}’

sequence   ::=  step { ’;’ step } *

step       ::=  [ decl_lst ] stmnt

one_decl   ::=  [ TYPE ivar { ’,’ ivar } * ]

decl_lst   ::=  one_decl { ’;’ one_decl } *

ivar       ::=  var_dcl | var_dcl ASGN expr | var_dcl ASGN ch_init

ch_init    ::=  ’[’ CONST ’]’ OF ’{’ TYPE { ’,’ TYPE } * ’}’

var_dcl    ::=  NAME [ ’[’ CONST ’]’ ]

var_ref    ::=  NAME [ ’[’ expr ’]’ ]

stmnt      ::=  var_ref ASGN expr
           | var_ref RCV margs
           | var_ref SND margs
           | PRINT ’(’ STRING { ’,’ expr } * ’)’
           | ASSERT expr
           | GOTO NAME
           | expr
           | NAME ’:’ stmnt
           | IF options FI
           | DO options OD
           | BREAK
           | ATOMIC ’{’ sequence ’}’

options    ::=  { SEP sequence } +

binop      ::=  ’+’ | ’-’ | ’*’ | ’/’ | ’%’ | ’&’ | ’|’ | ’>’ | ’<’
           | GE  | LE  | EQ  | NE  | AND | OR  | LSHIFT | RSHIFT

unop       ::=  ’˜’ | ’-’ | SND

expr       ::= ’(’ expr ’)’
           | expr binop expr
           | unop expr
           | RUN NAME ’(’ [ arg_lst ] ’)’
           | LEN ’(’ var_ref ’)’
           | var_ref RCV ’[’ margs ’]’
           | var_ref
           | CONST
           | TIMEOUT
           | var_ref ’.’ var_ref
           | var_ref ’:’ NAME
```

```
arg_lst    ::=  expr { ’,’ expr } *

margs      ::=  arg_lst | expr ’(’ arg_lst ’)’
```