Low-Fat Recipes for Spin

Theo C. Ruys

Faculty of Computer Science, University of Twente.
P.O. Box 217, 7500 AE Enschede, The Netherlands.
ruys@cs.utwente.nl

Abstract Since the introduction of the first version of the model checker Spin in 1991, many papers have been written on improvements to the tool and on industrial applications of the tool. Less attention has been given to the pragmatic use of Spin. This paper presents several techniques to optimise both the modelling and verification activities when using Spin.

Introduction

Since the introduction of the first version of the model checker Spin [5] in 1991 (accompanying Gerard Holzmann's book [6] on Spin), many papers have been published on technical improvements to Spin. The extensive list of industrial applications [8] shows that Spin has already been proven quite useful. The proceedings of the Spin Workshops [18] give a good overview of the (applied) research on Spin. It is surprising that less attention has been given to the pragmatic use of Spin; there is not even a Frequently Asked Questions (FAQ) list for Spin.

With respect to verification tools that need extensive user guidance – like theorem provers and proof checkers – model checkers are often put forward as 'press-on-the-button' tools: given a model and a property, pressing the 'verify' button of the model checker is enough for the tool to prove or disprove the property. If both the model and the property are readily available, this claim might be true. However, the formalization of both the model and the properties is usually not a trivial task. Furthermore, due to the infamous state space explosion problem, both the model and the property to be verified should be coded as efficient as possible for the model checker that is being used.

Now that model checking tools in general and Spin in particular are becoming more widespread in use [7], these tools are starting to be adopted by people that only want to press the button and that do not know precisely what is 'under the hood' of such verification tools. During technology transfer projects and during the education of our students we experienced that – without proper guidance – Promela and Spin are not being used in the most optimal way. On the one hand, Promela, because it resembles `C` [9], is regarded as a high level programming language. On the other hand, Spin is seen as a magic tool that can verify even the largest systems. Promela models are being constructed as some sort of `C` programs that may be good specifications and functional models, but may not be as efficient to verify. Several solutions to this potential misuse of model checkers come to mind:

- *More agressive tools.* If model checking tools would mimic optimizing compilers more closely, all non-efficiency could be optimised away. The limited price to pay would be a drop in compilation and runtime speed.
- *Restrict the language.* If the user of the model checker cannot use 'expensive' constructs, the model will be efficient by construction. The catch here is that in general it will be more difficult to model systems.
- *Educate the users.* If the users know what constructs should be avoided and what data and control structures are most efficient, the user can improve his modelling skills himself.

This paper aims to be helpful with respect to the last solution. We present some (shortened) selected techniques from "Effective SPIN" [15], a collection of proven best practices when using SPIN.

Effective Modelling and Verification Press-on-the-button verification is only feasible for small to medium sized applications. Industrial size applications need aggressive use of the modelling language, the properties to be checked and the verification tool itself. As discussed above, there is generally a big difference in efficiency in the models developed by a 'casual' user and the models developed by an 'expert' user. Moreover, the 'expert' user knows how to exploit the directives and options of the model checker to optimise the verification runs. Efficient use of model checking tools seems to require an 'assembler programming' approach to model building: use all tricks of the model checker to minimise the state space of the model and make the verification process as efficient as possible. The 'expert' verification engineer resembles the seasoned programmer, who not only has a deep knowledge and understanding of data structures and algorithms but also knows the options and directives to tune the programming tools that he is using.

With model checking tools there is – just as with programming – a trade-off between time and space requirements. For the model checking process, however, the space requirements are much more important than the time requirements. Because of the state space explosion, it is crucial to reduce the number of states as much as possible. So reduction of the number of states is the first consideration. The minimization of the size of the state vector (i.e. the amount of memory which is needed to encode a single state) is the next concern. Only in the last case, reduction of the verification time should be taken into account. SPIN has several options and directives to tune the verification process. Not surprisingly, many of these options are related to the trade-off between space and time requirements. An efficient verification model is usually optimised and tuned towards the property that has to be verified. It is not unusual to have a different model for each different property. This differs from programming, where one usually has only a single program (and older superseded versions of that program).

This paper presents a couple of 'expert' techniques to optimise both the modelling and verification process when using SPIN.[1] These techniqes are mostly

[1] To determine whether one is already on the 'expert' level of PROMELA one could do the following: start XSPIN, press "Help" and read the last section "Reducing

concerned with the minimization of the number of states or the reduction of the state vector. The techniques discussed here are not answers to FAQs, but are more like 'recipes from a cookbook' in the style of [2] and [12]. The recipes are presented in a tutorial oriented manner. When appropriate, there will be pointers to more formal and technical discussions, though. This paper is intended to be useful for intermediate to advanced users of SPIN; [15] has more sections for novice users of SPIN.

All proposed techniques can be used with standard PROMELA and standard SPIN. No modifications to SPIN are needed. Some hints and tips will be trivial to experienced C/C++ programmers, but might be 'eye-openers' for SPIN users that originate from the 'formal methods community'. Several of the tips and techniques presented in this report, may even look like terrible 'hacks': horrible and unreadable deviations from the original specification or model.[2]

We hope that the recipes will not only be adopted as efficient ways to achieve specific goals in PROMELA or SPIN, but also induce a new way of thinking about the verification process with SPIN.

Experiments The techniques discussed in this paper and the advice given are verified by numerous experiments with SPIN itself. Summaries of the results of (most of) these experiments are included in the report. All verification experiments are run on a Dell Inspiron 7000 Notebook computer running Red Hat Linux 6.1 (Linux kernel version 2.2.12). The Dell computer is driven by a Pentium II/300Mhz with 128Mb of main memory. For the `pan` verification runs we limited the memory to 64Mb though. For our experiments we used SPIN version 3.3.10.

Some verification runs have been repeated with certain optimization settings enabled or disabled. The different types of verification runs are identified as follows:

`default`	default XSPIN settings
`-o3`	disables SPIN's smart merging of safe sequences
`-DNOREDUCE`	disables partial order reduction
`-DCOLLAPSE`	a state vector compression mode
`-DMA=n`	uses a minimised DFA encoding for the state space

See [19] for details on these verification options.

Literate Programming The recipes in the paper are illustrated by PROMELA code fragments. These PROMELA fragments are presented as *literate programs* [10, 14]. Literate programming is the act of writing computer programs primarily as documents to be read by human beings, and only secondarily as instructions

Complexity". The SPIN user who already lives by all these rules of thumb, is on the right track.

[2] Some of the techniques discussed in this paper should probably be done by SPIN instead of the user. Future versions of SPIN might incorporate optimizations (e.g. assignment of arrays, *efficient* arrays of `bits`, checking for pure atomicity, etc.) which would make the discussions in this paper obsolete. Until then, one has to adopt these techniques manually.

to be executed by computers [14]. A literate program combines source code and documentation in a single file. Literate programming *tools* then parse such a file to produce either readable documentation or compilable source code.

We use `noweb`, developed by Norman Ramsey as our literate programming tool. `noweb` [14, 13] is similar to Knuth's `WEB`, only simpler. Unlike `WEB`, `noweb` is independent of the programming language to be literated.

We briefly introduce the reader to the `noweb` style of literate programming. A literate document consists of *code chunks* and *document chunks*. What follows is a code chunk.

1 ⟨*sample code chunk* 1⟩≡
```
proctype Silly()
{
  ⟨Silly's body 2⟩
}
```

In this code fragment, the chunk ⟨*sample code chunk* 1⟩ is defined. In the left margin, `noweb` shows the unique number of the *code chunk*.[3] When the name of a code chunk appears in the definition of another code chunk, it stands for the corresponding replacement text. In our simple example, ⟨*sample code chunk* 1⟩ uses the code chunk ⟨*Silly's body* 2⟩, which is defined as follows.

2 ⟨*Silly's body* 2⟩≡ (1) 3▷
```
do
:: skip
od ;
```

In the right margin of the definition of a chunk, between parenthesis, the tags of the code chunks that *use* this particular code chunk are listed. In this case, this list only contains the tag of ⟨*sample code chunk* 1⟩. It's possible and common practice to give the same name to several different code chunks. Continuing our example, we can expand our `Silly` process as follows.

3 ⟨*Silly's body* 2⟩+≡ (1) ◁2
```
assert(0) ;
```

The +≡ here indicates that the code chunk ⟨*Silly's body*⟩ has appeared before. The PROMELA code following +≡ will be appended to the previous replacement text for the same name. When such continuations of code chunk definitions are used, `noweb` provides more information in the right margin; it indicates the previous definition (◁) and the next definition (▷) of the same code chunk.

Recipe 1 – Macros, `inline` definitions and `m4`

Unlike most programming languages, PROMELA does not support the concept of procedures or functions to structure a PROMELA model. Instead, PROMELA offers the macro mechanism of the `cpp` preprocessor [9] and – since SPIN version 3.2.0

[3] In this paper the `WEB` style of chunk numbering is used. Another popular way of chunk identification is a tag *page.n*, where *n* indicates the *n*-th chunk on page *page*.

– the semantically equivalent `inline` construct. As the name of the construct already suggests, an invocation of an `inline` definition amounts to the automatic inlining of the text of the `inline` definition into the body of the process that invokes it.

Although limited with respect to native procedures or functions, `inline` definitions can still be quite helpful to structure a PROMELA model. The `cpp` macro mechanism is convenient for defining constants and symbolic propositions (e.g. XSPIN's dialog window for LTL verification runs). Furthermore, the `cpp` preprocessor can be used to parameterise a PROMELA model.

Note that SPIN's on-line documentation [19] suggests a third method to simulate `procedures`. A seperate server process needs to be declared with the body of the procedure. This server process responds to the user processes via a special globally defined channel, and responding to these requests via an user provided local channel. In the light of our 'assembler modelling' approach it will be clear that this method is rejected for its inefficiency.

In this section, we discuss some common `cpp` macros and `inline` tricks that have proven useful within the context of PROMELA models. We will also show some of the limitations of `inline` definitions with respect to parameterising PROMELA models and introduce the reader to `m4`, a more powerful macro processor than `cpp`.

1.1 Some `cpp` Macros

To get a feeling of `cpp` macros, we first introduce some useful `cpp` one-liners. These macros will be used in other parts of this report as well.

`IF/FI` PROMELA does not support a pure deterministic conditional statement, To model a deterministic conditional one has to reside to the following construct:

```
if
:: ⟨boolean expression⟩ -> ⟨then part⟩
:: else -> ⟨else part⟩
fi ;
```

If the ⟨*else part*⟩ is missing (i.e. equal to `skip`), the construct becomes a rather clumsy way to model the equivalent of the following piece of `C` code:

```
if ( ⟨boolean expression⟩ ) {
 ⟨then part⟩
}
```

The following two macros `IF` and `FI` can be used as a convenient shorthand for a deterministic conditional:

4 ⟨*cpp macros 4*⟩≡ 5 ▷

```
#define IF if ::
#define FI :: else fi
```

Now we can write:

```
IF ⟨boolean expression⟩ -> ⟨then part⟩
FI;
```

`IMPLIES` When checking properties in an `assert` statement, it often happens that one needs to check a logical implication: $p \Rightarrow q$. The $\Rightarrow$ operator does not have a direct counterpart in PROMELA. Instead we encode the equivalent $\neg p \vee q$ as a `cpp` macro:

5 ⟨*cpp macros* 4⟩+≡ ◁4

```
#define IMPLIES(p,q) ((!p) || (q))
```

1.2 A Poor Man's Assignment

Although PROMELA supports `array`s and `typedef` variables, these structured types are not (yet) 'first class citizens' of the language. For example, it is not possible to use PROMELA's assignment statement (i.e. '`=`') to copy one `array` or `typedef` variable to another.[4] Here, the `cpp` macro or `inline` construct can be helpful to implement a "poor man's assignment" or copy procedure. As an example, suppose a PROMELA model contains the following `typedef` definition:

6 ⟨*typedef Foo* 6⟩≡

```
typedef Foo {
  byte b1 ;
  byte b2 ;
} ;
```

The `inline` definition to copy one `Foo` variable to another is now trivial:

7 ⟨*inline CopyFoo* 7⟩≡

```
inline CopyFoo(src,dest)
{
  d_step {
    dest.b1 = src.b1 ;
    dest.b2 = src.b2 ;
  }
}
```

[4] If one tries to assign a complete `typedef` variable, SPIN will issue an 'incomplete structure ref' error. But beware: if one tries to assign a complete `array` variable, SPIN will *not* complain. SPIN even allows assignment of incompatible arrays (i.e. different base type or different number of elements). But instead of copying the complete `array`, SPIN will only copy the first element of the `array`. The reason for this is that the name of an `array` variable is treated as an alias to the first element of the particular `array`.
Please note that PROMELA does allow initialisation of a complete array in the declaration of the array, though. The declaration
⟨*type*⟩ `a[N]=val ;`
initialises all `N` elements of `a` to `val`.

Similarly, one can use the following `inline` definition to assign the value `val` to the elements of an array `a` of length `n`.

8 ⟨*inline AssignArray* 8⟩≡

```
inline AssignArray(a,n,val)
{
  byte i ;
  d_step {
    i=0 ;
    do
    ::  i <  n -> a[i] = val ; i++
    ::  i >= n -> break
    od ;
    i=0 ;
  }
}
```

Note that the variable `i` is *not* local to the `inline` definition, but instead will be a local variable in all processes that invoke the `AssignArray` definition. To make sure that the overhead of the local variable is kept to a minimum, the variable `i` is reset to `0` at the end of the `d_step`. In this way, system states will not differ on the basis of the temporary variable `i`. See Recipe 5 for details on this idiom.

It would even have been more efficient if we would be able to 'hide' the variable `i` from the state vector using the PROMELA keyword `hidden`. Unfortunately, the current version of SPIN only allows global variables to be declared as `hidden` variables. So in order to hide `i`, we should declare `i` as a global variable and remove the declaration of `i` from the `inline` macro.

Note that one has to supply `inline` definitions for all `typedef` objects or `array` variables that have to copied or initialised.

1.3 Parameterised Protocols

Communication protocols are often parameterised by some symbolic constants. Typical parameters are the number of processes, the length of the communication buffers, the window size of the protocol, etc. When modelling such a parameterised protocol in PROMELA one usually uses the macro mechanism of the preprocessor `cpp` to define the parameters at the start of the PROMELA model. For example, we could use the following PROMELA fragment

```
#define N       3
#define WSIZE   4
#define CL      2
```

to specify `N` protocol instances, a windowsize of `WSIZE` and a channel length of `CL` elements.

Each time the PROMELA model has to be validated with different values of the parameters, the constants need to be changed explicitly in the PROMELA model. To really make the constants parameters to the PROMELA model, SPIN

provides the preprocessor related options `-D` and `-E` to move the definition of such parameters outside the PROMELA model:

`-Dyyy` pass `-Dyyy` to the preprocessor

`-Eyyy` pass `yyy` to the preprocessor

Instead of defining the parameters in the PROMELA model itself, one can run SPIN as follows:

`spin -DN=3 -DWSIZE=4 -DCL=2` ⟨*promela file*⟩

Consequently, the ⟨*promela file*⟩ does not have to be changed for different values of the protocol parameters. When parameters are set in this way using the command-line, it is recommended to specify default values for the parameters in the PROMELA model itself. For example:

```
#ifndef N
#define N 3
#endif
```

In practice, changing one of the parameters of a PROMELA model often means that some other statements have to be altered as well. For example, consider the following PROMELA fragment, where a message `MSG` is non-deterministicly sent to one of the available `N` processes. We assume that the sending over the channels `to[i]` cannot be blocked.

9 ⟨*non-deterministic send* - `if` 9⟩≡

```
if
::  to[0] ! MSG
::  to[1] ! MSG
::  to[2] ! MSG
::  to[3] ! MSG
fi ;
```

In this case `N` is equal to `4`. The number of processes parameter is hardcoded into the model; if `N` is changed from `4` to `7`, we have to add three more lines. We could make the sending of the `MSG` depend on `N` using PROMELA's `do` statement:[5]

10 ⟨*non-deterministic send* - `do` 10⟩≡

```
byte i ;
atomic {
  i=0 ;
  do
  ::  i<N-1  -> i++
  ::  i<N-1  -> to[i]   ! MSG ; break
  ::  i>=N-1 -> to[N-1] ! MSG ; break
  od ;
  i=0 ;
}
```

This `do`-solution is less efficient than the straightforward `if` clause: not only do we need an extra variable to loop through the possible processes `0..N`, the `do`-construct also performs worse with respect to the execution time and the search

[5] Note that the `do`-solution is only semantically equivalent to the `if`-solution if the sending over the channels `to[i]` cannot be blocked.

cpp macro	m4 macro
`#define MAX 5`	`define(‘MAX’,‘5’)`
`#define P (a>5 && b<10)`	`define(‘P’,‘a>5 && b<10’)`
`#define IMPLIES(x,y) ((!x) || (y))`	`define(‘IMPLIES’,‘((!$1) || ($2))’)`
`#include "filename"`	`include(‘filename’)`

Table 1. Some `cpp` macros and their `m4` counterparts

depth.[6] Moreover, the `do`-solution is clearly less readable than the original `if` construct. The only advantage of the `do`-solution is that it is parameterised in `N`.

Unfortunately enough, the `cpp` preprocessor is not expressive enough to let a macro expand to the `if` solution: `cpp` does not have a looping construct that depends on some numeric constant. A more powerful preprocessor is needed.

1.4 The `m4` Macro Processor

Like `cpp`, `m4` [16] is a macro processor in the sense that it copies its input to the output, expanding macros as it goes. `m4` is being used either as a front-end to compilers, or as a macro processor in its own right. `m4` is much more powerful and flexible than `cpp`. `m4` is widely available on all `UNIX`es.[7] In the context of PROMELA and SPIN, `m4` has turned out to be valuable tool for making PROMELA models more generic without losing efficiency.

The use of a different preprocessor than SPIN's is anticipated in SPIN with the `-Pxxx` option. To make SPIN use `m4` instead of `cpp`, one simply issues the command

```
spin -Pm4 -E-s
```
[8]

This report is not the place to describe `m4` in great detail. The interested reader should refer to [16] instead. We will only briefly discuss some differences between `cpp` and `m4` to make a migration to `m4` easier. And of course we will present the parameterised `m4` macro that expands to the ⟨*non-deterministic send* - `if` 9⟩ chunk of the previous section.

Table 1 shows some `cpp` macros and their `m4` equivalent counterparts. The `m4` macro processor uses *quoted strings* (i.e. a string between the characters ‘ and ’) to specify the arguments of the `define` macros. Naturally, `m4` also provides constructs to conditionally include or exclude some program fragments. For example, the `cpp` construct

[6] In Recipe 3 “Randomness” we discuss the differences between the `if` and `do` solutions in greater detail.

[7] A warning on `m4` from [16]: “Some people found `m4` to be fairly addictive. ... Beware that `m4` can be dangerous for the health of compulsive programmers.”

[8] The option `-s` which is passed to `m4` using SPIN's `-E` option, is needed to ensure the correct synchronisation of line numbers and file names within the PROMELA source file(s). See [16] for details.

`#ifdef name`
⟨*then ...*⟩
`#else`
⟨*else ...*⟩
`#endif`
	has the following `m4` counterpart:	`ifdef(‘name’,‘`
		⟨*then ...*⟩ `’,‘`
		⟨*else ...*⟩ `’)`

And the `if`, the `ifndef` and `undef` constructs of `cpp` have equivalent commands within `m4` as well.

Comments in `m4` input files are normally delimited by the characters ‘`#`’ and a newline character. These comment delimiters can be changed to any string, using `m4`’s built-in macro `changecom`. To retain PROMELA style comments – i.e. the `cpp` style comments – we change the comment delimiters to `/*` and `*/`.

11 ⟨*m4 macros* 11⟩≡ 12 ▷
```
changecom(‘/*’,‘*/’)
```

Besides counterparts for all `cpp` commands, `m4` supports several additional preprocessing features. To implement a general looping construct, only the stack-like redefinition macros, the recursion construct and the integer arithmetic operations of `m4` are needed, though. The following `forloop` macro is from [16]:

12 ⟨*m4 macros* 11⟩+≡ ◁ 11
```
define(‘forloop’,
       ‘pushdef(‘$1’, ‘$2’)_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)popdef(‘$1’)’)
define(‘_forloop’,
       ‘$4’’ifelse($1, ‘$3’, ,
       ‘define(‘$1’, incr($1))_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)’)’)
```

Understanding the implementation of the `forloop` macro is not really needed, only the result of an invocation of `forloop` is important here. The `forloop` macro expects 4 parameters. The first parameter is the looping variable. The second and third parameter are the start and end value of the looping variable, respectively. The last argument of `forall` is the string that should be written for each value of the looping variable.
For example, the macro invocation
```
forloop(‘i’, 1, 8, ‘i ’)
```
expands to
```
1 2 3 4 5 6 7 8
```
For each value of `i`, the string ‘`i `’ is written, with the actual value substituted for `i`. Using `forloop`, we now are able to write a parameterised version of ⟨*non-deterministic send* - `if` 9⟩:

13 ⟨*non-deterministic send* - `if` *using* `m4` 13⟩≡
```
if
forloop(‘i’, 0, eval(N-1), ‘:: to[i] ! MSG
’)fi ;
```

If `N` has been defined to be 4, this chunk 13 will expand exactly to chunk 9. Although we managed to parameterise the guards of the `if` clause, the `m4` construct using `forloop` is clearly less readable than both the original static `if` clause and the `do` solution. In Recipe 3, though, we will show that the `m4` construct proves to be an efficient and parameterised solution.

The `forloop` macro can also be quite useful in a (`boolean`) expression. Suppose that a process waits for all `N` processes to become ready, i.e. the boolean `ready[i]` is `true` for all `i` ∈ `0..N-1`:

```
ready[0] && ready[1] && ... && ready[N-1]
```

Again, we cannot use `cpp` to build this expression which depends on the parameter `N`. The `forloop` macro is straightforward, however:

```
forloop('i', 0, eval(N-1), 'ready[i] &&') true
```

Conclusions To structure a PROMELA model or to parameterise the model, the use of `cpp` and `inline` constructs usually suffices. The power user, however, might consider to add `m4` to his (verification) toolbox: this macro processor is more powerful than `cpp` and the parameterising of the PROMELA model is usually more elegantly. Moreover – as we will see in Recipe 3 – the resulting PROMELA model can be more efficient in terms of the number of states or the needed search depth. It is possible to mix `m4` and `cpp` constructs in a single PROMELA model. This is not considered good practice, though.

As a last remark, remember that PROMELA is a protocol *meta language*, not a programming language. Shorthands like `IF` and `forloop` should be used with caution.

Recipe 2 – Atomicity

This section discusses issues related to the `atomic` and `d_step` constructs of PROMELA, which both introduce a sequence of statements that is to be executed indivisibly. The `atomic` sequence is allowed to contain non-deterministic choices, whereas the `d_step` (i.e. the deterministic step) may only contain deterministic statements.[9]

Both constructs can be used to reduce the complexity of the validation model and to improve the efficiency of verification runs [19]. As we will see in Recipe 5, the `d_step` construct can also be seen as a mechanism to define an indivisible statement in the language at the user-level, and thus to extend the semantics of PROMELA itself [20].

The semantics of the `atomic` clause of PROMELA has changed over the years. In version 1.x of SPIN, it would causes a run-time error if any statement, other than the first statement, blocks in an `atomic` sequence [17]. However, since version 2 of SPIN, it is legal for an `atomic` sequence to block. If any statement within the atomic sequence blocks, the atomicity is lost, and other processes are allowed to execute arbitrarily many statements [20]. The `d_step` is not allowed to block, though. The `pan` verifier will abort the verification if it detects a blocking `d_step`.

[9] Although the `d_step` clause should only contain 'deterministic' statements, the `pan` verifier will not check this. If the verifier encounters a non-deterministic choice, it will just choose the first alternative.

2.1 Atomicity of Single Statements

Before turning to the `atomic` and `d_step` constructs themselves, we briefly discuss the granularity of PROMELA statements. In the realm of (competing) parallel processes that share global memory, the implementation of computations and assignment statements is liable to result in incorrect behaviour due to race conditions between the processes. The abstraction level of PROMELA is higher: a single statement is assumed to be indivisible. For example, an assignment like

```
a = b+c*d-e*f ;
```

is considered to be atomic within PROMELA. If one wants to check the correctness of a possible low-level implementation of such an assignment, one should manually split up the assignment using temporary variables. For example:

```
x1 = c*d ;
x2 = e*f ;
x1 = b+x1 ;
a  = x1-x2 ;
```

To exclude the temporary variables `x1` and `x2` from the state vector, they should be defined as `hidden` global variables.

2.2 `atomic` is not always atomic

New users often expect the version 1.x semantics of SPIN: placing an `atomic` clause around a sequence of statements should *ensure* the atomicity of the statements. This erroneous perspective can lead to unexpected errors that are hard to find. For example, suppose a PROMELA model contains an `atomic` clause which is assumed to be indivisible. During a verification run, however, the `atomic` clause blocks and control is passed to one of the other processes. Later during the search, the property that is being checked is violated, due to the premature ending of the `atomic` clause. Still assuming the atomicity of the clause, the user does not understand why the property has been violated.

The statement that causes an `atomic` clause to block is often an `if` or `do` statement. If the set of guards of the `if` or `do` statement is not complete and the `else` statement is missing, the particular statement might block. This error can easily slip into the model, when behaviour is added to the PROMELA model (i.e. new possible values for variables, new `mtype` messages that are sent over a channel), that is not anticipated by the particular `if` or `do` statement.

Although the semantics of the `atomic` clause have changed, it is still relatively easy to check whether `atomic` clauses are 'pure' in the sense that they are not exited prematuraly due to a blocking statement. The following steps are sufficient:

- declare a *global* `bit` variable `aflag`;
- set `aflag` to `1` on entrance of each `atomic` block that has to be checked for 'pure atomicity': immediately after the first statement or guard of the `atomic` block;

– set `aflag` to 0 on leaving those `atomic` blocks: immediately before the closing '`}`' of the block;
– use SPIN to verify that `aflag` is always equal to zero, i.e. verify that the invariant property `[]P` holds, where `P` is equal to `!aflag`.

For example, to verify the following `atomic` clause

14 ⟨*atomic block* 14⟩≡
```
atomic {
  guard ;
  ...
}
```

for 'pure atomicity', it would have to be changed to

15 ⟨*atomic block with* `aflag` 15⟩≡
```
atomic {
  guard ;
  aflag=1 ;
  ...
  aflag=0 ;
}
```

A drawback of this method is that all `atomic` clauses have to be altered in the PROMELA model to check for 'pure atomicity'. Instead, we could also use `cpp` macros such that the checking can be done conditionally. The ⟨*aflag declarations* 16⟩ chunk below defines the necessary macros:

16 ⟨*aflag declarations* 16⟩≡
```
#ifdef CHECK_ATOMICITY
bit aflag ;
#define SET_AFLAG       aflag=1
#define RESET_AFLAG     aflag=0
#else
#define SET_AFLAG       skip
#define RESET_AFLAG     skip
#endif
```

The `SET_AFLAG` and `RESET_AFLAG` macros are only 'active' when `CHECK_ATOMICITY` is defined. The ⟨*atomic block with* `aflag` 15⟩ fragment can now be changed to:

17 ⟨*atomic block with* `AFLAG` *macros* 17⟩≡
```
atomic {
  guard ;
  SET_AFLAG ;
  ...
  RESET_AFLAG ;
}
```

2.3 Infinity and Atomicity

Most reactive systems – like communication protocols – execute forever. Spin does not have problems verifying such systems with infinite behaviour.[10] Infinity and the `atomic` and `d_step` constructs do not mix, though. Consider the following trivial infinite loop in PROMELA:

18 ⟨*infinite loop* 18⟩≡ (19–21)

```
bit b ;
do
::  b=1-b
od ;
```

which is encapsulated in the following `proctype`:

19 ⟨*infinite-normal.pr* 19⟩≡

```
active proctype Infinite()
{
  ⟨infinite loop 18⟩
}
```

When this PROMELA model is checked (e.g. for invalid endstates), `pan` will terminate normally and report that 2 states are stored with a maximum search depth of 1.
Things change, however, if we enclose the ⟨*infinite loop* 18⟩ into an `atomic` clause:

20 ⟨*infinite-atomic.pr* 20⟩≡

```
active proctype Infinite()
{
  atomic { ⟨infinite loop 18⟩ }
}
```

There will still be only one state, but `pan` cannot 'get out' of the `atomic` loop; the `pan` verifier will continue to execute the assignment statement. Luckily, `pan` will (eventually) complain that the search depth was too small: every execution of the assignment will have been put onto the stack. When we enclose the ⟨*infinite loop* 18⟩ in a `d_step`, we get into trouble, though.

21 ⟨*infinite-d_step.pr* 21⟩≡

```
active proctype Infinite()
{
  d_step { ⟨infinite loop 18⟩ }
}
```

The `pan` verifier will never be able to complete its `d_step` transition and will keep executing the assignment statement; all in a single search step. The verifier will get into an endless (livelock) loop and will never allow one of the other processes to proceed. This error is not easy to spot as it seems as if `pan` is very busy traversing the state space.

[10] Note that PROMELA models always define finite state systems. Thus infinite behaviour in PROMELA involves looping: visiting a state that has already been visited before.

Although this example is trivial, an endless loop in a `d_step` clause is not unlikely to occur in practice. For example, consider the `inline` definition of ⟨*inline AssignArray* 8⟩, which initialises an array `a` of length `n`. The increment statement `i++` in the `do`-loop can easily be forgotten.

So before putting a computation into a `d_step` one should make sure that the computation does not contain an infinite loop. A simple way to check this is to first enclose the computation sequence into an `atomic` clause. If the maximum search depth turns out to be too small due to the `atomic` clause, the clause probably contains an infinite loop.

Conclusions In this recipe we discussed the 'atomicity' constructs of PROMELA. We have shown how to check for 'pure atomicity' when using the `atomic` clause. Furthermore, we discussed the pitfalls regarding infinite behaviour in combination with `atomic` and `d_step`.

Recipe 3 – Randomness

The file rand.html from [19] mentions the following: "There is no random number generation function in PROMELA. ... In almost all cases, PROMELA's notion of non-determinism can replace the need for a random number generator." In general this is true. Randomness is a concept used in program *implementation* (e.g. in simulutation and testing), whereas non-determinism is a concept used in the *specification* of systems and hence, in model checking and verification. An attempt to construct a random generator in PROMELA often reflects a misunderstanding of the user with respect to the model. For verification, in general, only specific possibilities (e.g. boundary values, valid and invalid choices) need modelling.

Still, there sometimes seems to be a need for an explicit randomise construct. Especially users new to SPIN and less familiar with non-deterministic choices, expect a random number generation function in PROMELA. Furthermore, in the initial phase of the modelling of a system, an explicit random construct can be quite useful.

In this section we investigate and compare several possibilities to add a `random` definition to the PROMELA language. The randomness example proves to be a nice example to get a feeling of the "assembler programming" approach to model building.

`do` solution A natural first attempt to an explicit randomise construct – which is commonly seen – is the following piece PROMELA definition:

22 ⟨*inline: random - plain* `do` *1st try* 22⟩≡

```
inline random(i,N)
{
  ⟨random - do 1st try⟩
}
```

where the body of ⟨*random* - `do` *1st try* 23⟩ could be defined as:

23 ⟨*random* - `do` *(1st attempt)* 23⟩≡

```
i=0 ;
do
:: i<N  -> i++
:: i<N  -> break
:: i>=N -> break
od ;
```

The `do`-loop is used to non-deterministically increment the variable `i` or to break out of the loop. After the loop the variable `i` will have a value from the range `0..N`. We see that for both guards `i<N` and `i>=N` we can always `break` out of the loop. So, an elegant and slightly more efficient randomise construct is the following:

24 ⟨*inline: random* - *plain* `do` 24⟩≡

```
inline random(i,N)
{
  ⟨random - do 25⟩
}
```

where ⟨*random* - `do` 25⟩ is defined as:

25 ⟨*random* - `do` 25⟩≡ (24 26)

```
i=0 ;
do
:: i<N -> i++
:: break
od ;
```

The construction can improved even further by placing the complete ⟨*random do*⟩ chunk in an `atomic` clause.[11]

26 ⟨*inline: random* - `atomic do` 26⟩≡

```
inline random(i,N)
{
  atomic { ⟨random - do 25⟩ }
}
```

`if` solution Similar to ⟨*non-deterministic send* - `if` 9⟩, we can also use an `if`-clause to set the random value in a single transition. For example, if `N` is `4`, we could also set `i` to a random value between `0` and `N` using the following code:

```
if
:: i=0
:: i=1
:: i=2
:: i=3
:: i=4
fi ;
```

[11] Note that we cannot use a `d_step` clause here, because the random choice is based on the non-deterministic guards in the `do`-loop.

SPIN will non-deterministicly choose one of guards to set `i` to a value in the range `0..4`. The drawback of the `if` solution is that the code chunk has to be altered when the constant `N` is changed. As explained in Recipe 1, we cannot use the `cpp` preprocessor to let a macro expand dynamically to a variable number of guards, based on the parameter `N`. Instead we use the `m4` macro `forloop` defined in ⟨*m4 macros* 12⟩ to dynamically build the `if` clause:[12]

27 ⟨*inline: random* - `if` 27⟩≡

```
inline random(i)
{
  if
forloop(‘j’, 0, eval(N), ‘  :: i=j
’)  fi ;
}
```

Pseudo-random Generator Apart from the non-deterministic techniques that we discussed above, one can also model a deterministic, pseudo-random generator in PROMELA. For example, after defining the macro

```
#define RANDOM (seed*3 + 14) % 100
```

every subsequent assignment

```
seed = RANDOM ;
```

will set `seed` to a "pseudo-random" value between `0` and `99`. It will be clear that this is a different kind of randomness than the non-deterministic `do` and `if` techniques. In the remainder of this recipe, we will not discuss pseudo-random generators any further.

3.1 Comparison

To compare the different implementations of the `random` definition, we have run two types of test series with SPIN:

- We have verified a PROMELA model where `random(i,N)` was called only once with `N==50`.
- To check the verification time of the `random` construct, we also verified a PROMELA model where the `random(i,N)` definition was invoked `10000` times with `N==10`.

We distinguish between setting a local or global variable. The reason for making this distinction is that declaring variables to be local to a process *or* global to the complete model can have consequences on the effectiveness of the verification runs. Although semantically equal (unless the global variable is used in some other process), SPIN can optimise the use of local variables more aggressively because, by definition, a *local* variable will never be used by other processes. Thus, SPIN can savely apply live-variable analysis [1] on local variables within a process, whereas SPIN cannot do this for global variables.

[12] The only drawback of this `m4` approach is that we cannot make `N` a parameter of the `inline` definition; the value of `N` has to be known at macro expansion time.

implementation	options	depth reached	states stored	states matched	transitions
plain `do`	default	53	104	50	154
`atomic do`	default	55	104	50	154
`if`	default	2	53	50	103
plain `do`	`-o3`	104	205	50	255
`atomic do`	`-o3`	105	104	50	154
`if`	`-o3`	3	104	50	154

Table 2. local - `50`. Comparing different implementations to set a *local* variable to a random value between `0` and `50`.

implementation	options	depth reached	states stored	transitions	total memory (Mb)	time (sec)
plain `do`	default	130002	330003	430003	10.669	2.23
`atomic do`	default	150002	330003	1429900	11.149	11.05
`if`	default	20002	220003	1319900	6.390	6.21
plain `do`	`-o3`	250002	650003	750003	18.669	4.13
`atomic do`	`-o3`	260002	440003	1539900	15.770	14.07
`if`	`-o3`	40002	440003	1539900	10.250	6.69

Table 3. local - `10000x10`. Comparing different implementations to set a *local* variable `10000` times to a random value between `0` and `10`.

For the 'local variable' verification runs, for example, we used the following `Test` process:

28 ⟨*random-local-var.pr* 28⟩≡

```
active proctype Test()
{
  byte i ;
  random(i) ;
  assert((0<=i) && (i<=N)) ;
}
```

Furthermore, we repeated the verification runs with different optimization settings enabled and disabled. Enabling or disabling partial order reduction did not make any significant difference. On the other hand, disabling the "sequence merge mode" of SPIN (using the `-o3` option) gave different results. Tables 2-5 summarise the results of the various verification runs.

Local Table 2 and Table 3 list the results of randomly setting a local variable. The tables show that the plain `do` and `atomic do` solutions behave more or less the same for the default setting of SPIN. If the "sequence merge mode" is disabled with `-o3`, though, the plain `do` solution generates many more states than the `atomic do` construct. We also can conclude that using an `atomic` construct

implementation	options	depth reached	states stored	states matched	transitions
plain `do`	default	104	255	0	255
`atomic do`	default	55	154	0	154
`if`	default	3	154	0	154
plain `do`	`-o3`	104	255	0	255
`atomic do`	`-o3`	105	154	0	154
`if`	`-o3`	3	154	0	154

Table 4. global - `50`. Comparing different implementations to set a *global* variable to a random value between `0` and `50`.

implementation	options	depth reached	states stored	transitions	total memory (Mb)	time (sec)
plain `do`	default	250002	650013	750003	18.669	7.56
`atomic do`	default	160002	440013	1539900	13.130	18.35
`if`	default	40002	440013	1539900	10.250	10.52
plain `do`	`-o3`	250002	650013	750003	18.669	7.46
`atomic do`	`-o3`	260002	440013	1539900	15.770	20.64
`if`	`-o3`	40002	440013	1539900	10.250	10.71

Table 5. global - `10000x10`. Comparing different implementations to set a *global* variable `10000` times to a random value between `0` and `10`.

has negative influence on the running time of the verification: the `atomic do` solution is much slower than the plain `do` construct.

In the default setting, the `if` random solution behaves superior to both `do` solutions: the number of states is less and the depth of the `if` construct is constant, whereas the depth of both `do` solutions is linear in `N`. For `-o3`, the `if` solution results in as many states as the `atomic do` solution, but the search depth is still constant. The `if` solution is slightly slower than the plain `do` construct but a factor two faster than the `atomic do`.

Global Table 4 and Table 5 show the results of randomly setting a global variable. Now the number of states for the `if` and `atomic do` solutions are the same for both the default and `-o3` verification runs. Still, the depth of the `if` solution is superior. The plain `do` solution generates more states in both settings and only excels in its execution speed. Note that only the search depth of the `atomic do` construct is affected by changing the verification run from default to `-o3`.

Conclusions The `if` solution is favorable with respect to the number of states and the depth reached. For the same reason, even despite its fast running times, the *plain* `do` solution should be avoided. The `atomic do` solution suffers from the linear depth and the somewhat slower execution time. The advantage of the

`atomic do` solution is that it can be used for general `N` without changing the PROMELA source code. In the rare event, that you need an explicit randomise function, the `atomic do` will therefore suffice. For a general and efficient implementation, one should try the `m4` implementation of the `if` solution.

Recipe 4 – Array of `bits` - `bitvector`

PROMELA has borrowed the array mechanism of `C` to group related values into a single array variable. All PROMELA datatypes can be stored in an array. When modelling a system in PROMELA, an array of `bits` is quite useful to encode the (local) state of the system. For example, an array of `bits` can be used to model

- a collection of `on/off` switches of the system (e.g. a factory plant); or
- a set of processes in the system (e.g. in a multicast protocol)

Unfortunately, when using arrays of `bits`, SPIN will issue the following unnerving warning:

`spin: warning: bit-array` ⟨*array-name*⟩ `mapped to byte-array`

In other words, SPIN will allocate `8` times as much memory for the `bit` array in the state vector as was expected! In this section we will discuss a different way to encode arrays of `bits` in PROMELA, which is superior to SPIN's mapping to `byte`-arrays.

4.1 Bitvector

To implement an equivalent to PROMELA's array-mechanism, we define a library of `bitvectors`. A `bitvector` is an (unsigned) piece of memory, where each `bit` can be individually set, reset and tested. We use PROMELA's built-in integer types to represent the `bitvectors`: `byte` (max. 8 bits), `short` (max. 16 bits) and `int` (max. 32 bits). PROMELA also supports a variable length `unsigned` type (max. 8 bits). The following aspects of the various integer types have to be taken into account:

- The `byte` type is an *unsigned* type. The `short` and `int` types are *signed* integer types, which means that we have to be careful with the sign bit (left-most bit). Special care is needed in combination with logical shift operations to the right, because such operations also shift the signbit to the right.
- In numerical expressions, SPIN converts the operands to (32-bit) *signed* `int` values. Consequently, a bitvector `int` consisting of 32 ones (i.e. `~0`) is considered to be a negative `int` value. SPIN will generate a truncation warning when converting an `int` value back to an *unsigned* `byte`.
- Consequently, it is also not wise to use `-1` (i.e. 16 ones) for a `short`. In numerical expressions this value is converted to the `int` value `ffff`, which results in similar truncation warnings.

To be on the safe side, one should not use *signed* values (i.e. non-negative integers) to encode `bitvector`s, so:

- use `byte`s or `unsigned` variables for `bitvector`s with 2–8 `bit`s;
- use `short`s for `bitvector`s with 9–15 `bit`s;
- use `int`s for `bitvector`s with 16–31 `bit`s;

We define the following shorthands for `bitvector` declarations.

29 ⟨`bitvector` *macros* 29⟩≡ (34) 30▷
```
#define BITV_U(x,n)       unsigned x : n
#define BITV_8            byte
#define BITV_16           short
#define BITV_32           int
```

The suffixes `_8`, `_16` and `_32` indicate the number of bits the corresponding `bitvector` occupies.

30 ⟨`bitvector` *macros* 29⟩+≡ (34) ◁29 31▷
```
#define ALL_1S            2147483647
```

The constant `ALL_1S` is equal to 2^{31} and is represented by a zero followed by `31` ones. The constant `ALL_1S` is used to set all bits in a bitvector to `1`.

The `bit`s of a `bitvector` are manipulated using the logical *bitwise* operators of PROMELA: `~`, `&`, `|`, `^`, `<<` and `>>`. More details on these operators can be found in traditional textbooks on the programming language C (e.g. [9]). We define some basic operations to manipulate `bitvector`s. The following macros set the `i`-th bit of the `bitvector bv` to 0 and 1, respectively:

31 ⟨`bitvector` *macros* 29⟩+≡ (34) ◁30 32▷
```
#define SET_0(bv,i)       bv=bv&(~(1<<i))
#define SET_1(bv,i)       bv=bv|(1<<i)
```

We can also set all bits of a bitvector to `0` or `1` in a single instruction:

32 ⟨`bitvector` *macros* 29⟩+≡ (34) ◁31 33▷
```
#define SET_ALL_0(bv)     bv=0
#define SET_ALL_1(bv,n)   bv=ALL_1S>>(31-n) ;
```

The parameter `n` to `SET_ALL_1` denotes the number `bit`s to set to `1`. It seems natural to let `SET_ALL_1` just assign `ALL_1S` to `bv`. However, SPIN will issue a truncation error when `bv` is a `byte` or a `short`. The following two macros can be used to test whether the `i`-th `bit` is `0` or `1`, respectively:

33 ⟨`bitvector` *macros* 29⟩+≡ (34) ◁32
```
#define IS_0(bv,i)        (!(bv&(1<<i)))
#define IS_1(bv,i)        (bv&(1<<i))
```

The ⟨`bitvector` *macros*⟩ now provide the same functionality as the original array manipulation operations. Instead of writing `a[i]=1` one has to invoke the macro `SET_1(a,i)` and the boolean test `a[i]` now boils down to `IS_1(a,i)`.
All `bitvector` macros are stored in the 'header' file `bitvector.lpr`:

34 ⟨*bitvector.lpr* 34⟩≡
⟨`bitvector` *macros* 29⟩

Bigger bitvectors: revival of byte-arrays If one needs a bitvector with more than 31 individual bits, one can use an array of bytes to encode such a 'big' bitvector. The following macros again hide the implementation details from the user.

35 ⟨bitvector *macros using* byte-*arrays* 35⟩≡

```
#define BITV(bv,n)        byte bv[n]
#define SET_0(bv,i)       bv[i/8] = bv[i/8] & (~(1<<(i%8)))
#define SET_1(bv,i)       bv[i/8] = bv[i/8] | (1 << (i%8))
#define IS_0(bv,i)        (!(bv[i/8]&(1<<(i%8))))
#define IS_1(bv,i)        (bv[i/8]&(1<<(i%8)))
```

The array of bytes approach also becomes attractive if the number of bits that have to be stored is between 17 and 24; this would save one byte compared to the int implementation. A drawback of using an array of bytes to encode bitvectors is that manipulation of a complete bitvector is more problematic. For instance, testing whether all bits are equal to zero (or one) cannot be done using a cpp macro definition. In the remainder of this recipe we will only use bitvectors that are implemented by the simple data types: byte, short and int.

Note that encoding bitvectors using an array of bytes is as efficient as the implementation using the simple data types: byte, short and int. The results of the verification runs of bitvectors implemented by arrays of bytes can be found in [15].

4.2 Comparison

To compare Spin's byte-array implementation to the newly developed bitvector macros, we have written a simple Promela specification that models a *bridge* between two places A and B. At the start of the system, N persons are at A and they all have to cross the bridge to get to B.

The places A and B are modelled by Promela processes and the bridge itself is a (handshake) channel between A and B. The choice for the next person to cross the bridge is made non-deterministically.[13] We use the variable person to encode the presence of a person at either A or B. The variable person is either defined as a bit-array (and converted to a byte-array by Spin) or as as bitvector. If person[i] is 1 (or IS_1(person,i) is true) in process A it means that the i-th person is still at A.

To illustrate the usage of the bitvector operations, we include the definition of process B for the N=8 case:

36 ⟨*proctype B - one bridge* 36⟩≡

```
active proctype B()
{
  BITV_8  person ;
  byte    i ;
```

[13] Analogous to ⟨*inline: random* - if 27⟩, the 'random' choice for the next person to cross the bridge has been implemented by an if guard using m4.

implementation	options	state vector	depth reached	states stored	total memory (Mb)	time (sec)
byte-array	default	36	54	7702	2.507	0.07
bitvector	default	24	54	7702	2.404	0.09
byte-array	-DCOLLAPSE	36	54	7702	2.507	0.13
bitvector	-DCOLLAPSE	24	54	7702	2.507	0.11
byte-array	-DMA=60	36	54	7702	0.819	1.60
bitvector	-DMA=60	24	54	7702	0.512	1.98

Table 6. One bridge, 8 persons: byte-array vs. bitvector.

```
    SET_ALL_0(person) ;
    do
    :: (!ALL_HERE(8)) -> bridge ? i ; SET_1(person,i) ;
    :: else -> break
    od
  }
```

where ALL_HERE is defined as:

37 ⟨*bridge: macros* 37⟩≡

```
#define ALL_HERE(N)  ((person^((~0)<<N))==(~0))
```

We have verified three cases:

- One bridge, N=8. There is only one bridge between A and B. The number of persons at A is 8. The information stored in the person array can be coded in a single byte (i.e. BITV_8).
- One bridge, N=14. There is only one bridge between A and B. The number of persons at A is 14. Earlier experiments have shown that N=14 is the largest parameter for which the model can be verified exhaustively within 64Mb of memory.
- Two bridges, N=7. There are three processes A, B and C and there are two bridges: one between A and B and one between B and C. At the start there are 7 persons at A that have to go to C via B.

Spin's approach to map bit-arrays to byte-arrays may not be extremely problematic. If the state vector compression techniques of Spin would be able to compress the 7 extra zeros that are allocated for each bit in the byte-arrays, not much harm will be done. For that reason we have also verified the bridge models with two of Spin's advanced compressions methods enabled: -DCOLLAPSE and -DMA=60.

The results of the experiments are summarised in the Tables 6-8. The results show that the bitvector implementation indeed results in a much smaller state vector. Consequently, the verification of the bitvector models needs (much) less memory than Spin's byte-array implementation. Surprisingly enough, the

implementation	options	state vector	state vector - stored	state vector - overhead	depth reached	states stored	total memory (Mb)	time (sec)
byte-array	default	52	40	8	90	1204260	60.058	18.94
bitvector	default	28	16	8	90	1204260	31.180	15.78
byte-array	-DCOLLAPSE	52	24	12	90	1204260	45.209	32.63
bitvector	-DCOLLAPSE	28	18	12	90	1204260	38.143	28.43
byte-array	-DMA=60	52	-	-	90	1204260	66.358	456.63
bitvector	-DMA=60	28	-	-	90	1204260	36.865	454.29

Table 7. One bridge, 14 persons: byte-array vs. bitvector.

implementation	options	state vector	state vector - stored	state vector - overhead	depth reached	states stored	total memory (Mb)	time (sec)
byte-array	default	52	36	8	86	316379	16.127	5.07
bitvector	default	36	20	8	86	316379	11.109	4.62
byte-array	-DCOLLAPSE	52	10	12	86	316379	9.265	7.46
bitvector	-DCOLLAPSE	36	10	12	86	316379	9.163	7.01
byte-array	-DMA=60	52	-	-	86	316379	2.560	64.88
bitvector	-DMA=60	36	-	-	86	316379	1.638	72.92

Table 8. Two bridges, 7 persons: byte-array vs. bitvector.

bitvector implementation is also faster than the byte-array implementation (except for the -DMA=60 compression verification runs). The -DCOLLAPSE compression mode behaves spectacular on the byte-array verification runs, but only brings it closer to the bitvector implementation.

The bitvector implementation seems to be extremely efficient and difficult to compress any further. In the one bridge/N=14 case (Table 7), the default case of the bitvector run performs better than the two compressed verification runs.

Conclusions ¿From the results we conclude that there is no reason to stick to SPIN's byte-array implementation. With respect to state space considerations, the bitvector implementation is superior in all cases, including the compressed verification runs. Furthermore, the bitvector macros seem slightly faster than the array indexing implementation of SPIN.

Recipe 5 – Extending Promela - Deque

PROMELA is a protocol *modelling* language; it is not a specification language. One of the complaints about PROMELA that is often heard is that PROMELA resembles the programming language C [9] too much. The lack of more abstract datatypes than the built-in types bit, byte, array, etc., is seen as serious disadvantage. This view on PROMELA is not correct, though. The PROMELA language is rich

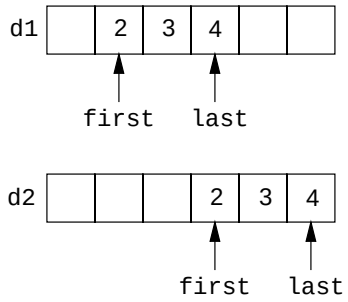

Figure 1. Erroneous implementation of a circular `deque` in PROMELA, using dynamic `first` and `last` pointers.

enough to add user-defined datatypes. The `typedef` construct can be used to define new datatypes and the `inline` or `cpp` macros (see Recipe 1) can be used to define operations on such new datatypes. If the bodies of the operations are enclosed in `d_step` clauses, the implementation will be highly efficient, as SPIN treats a complete `d_step` clause as a single transition.

The most important rule that should be followed when adding a new datatype `T` to PROMELA is that a value `t` of `T` is always represented by the same sequence of `bits`. The reason for this is that two states are considered equal by SPIN if the memory representation of both states is exactly the same. If the same value `t` can have several different memory footprints, SPIN will not be able to conclude that the same value `t` is used.

For example, suppose we would try to implement a double-ended `deque`-like datatype on top of the built-in `array` type of PROMELA. We treat the `array` as a circular buffer, using `first` and `last` pointers which point to the first element and last element of the `deque`, respectively. Such implementation allows the addition and removal of elements at both sides of the `deque`, which can be done quite efficiently. Figure 1 shows the `array` representations of `deque d1` and `d2` that are semantically equal. Both `deque`s contain the values `2`, `3` and `4`. However, apart from the fact that the `first` and `last` pointers of `d1` and `d2` are different, the array representation of `d1` and `d2` are clearly not equal. So, SPIN will treat the two `deque`s as being different.

5.1 Deque

To illustrate the power and elegance of PROMELA, we present a *correct* implementation of a double-ended `Deque` datatype in PROMELA. Elements can be added and removed from the front and the back of a `Deque` object. First we define the `Deque` datatype itself.

38 ⟨*typedef* `Deque` 38⟩≡

```
typedef Deque {
  byte  a[N] ;
  byte  length ;
  byte  i ;
} ;
```

The array `a` is used to hold the elements of the `Deque`. The array `a` can hold at most `N` elements of type `byte`. The field `length` holds the number of elements in the `deque`. The first element (i.e. the 'front') of a `Deque` variable will always be stored in `a[0]`, whereas the last element (i.e. the 'back') will reside in `a[length-1]`. Entries in `a` that are not used (i.e. `a[length..N-1]`) will always have value `0`.

The field `i` is only used as a temporary index variable within the array `a`. Outside of the `Deque` operations, it will always have the value `0`. Instead of having a *local* temporary field `i` for each `Deque`, one could also choose to use a single global variable for all `Deque` variables.[14] Having a single global temporary variable is more efficient with respect to the state vector, especially when the PROMELA model uses several `Deque` variables. The advantage of having the local `i` field in the `typedef` definition of `Deque`, though, is that the `typedef` definition is self-sufficient; the user can use the `typedef` definitions together with the `Deque` operations without having to declare additional variables. There is no danger for having nameclashes.

A clear disadvantage of the ⟨*typedef* `Deque` 38⟩ definition is that the type of the elements (i.e. `byte`) and the number of elements (i.e. `N`) are hardcoded into the `Deque` definition. If one needs another `Deque` in which the type of the elements or the maximum number of elements differs, one has to define a new `typedef` definition. To be more generic, we make both the type and the number of elements a parameter to the `Deque` datatype:

39 ⟨`Deque` *definition* 39⟩≡ (48) 42 ▷

```
#define DECLARE_DEQUE(T,MAX) \
typedef Deque_##T##MAX { \
  T     a[MAX] ; \
  byte  length ; \
  byte  i ; \
}
```

The `##` operation in the `typedef` is `cpp`'s string concatenation operator. Thus, the macro application `DECLARE_DEQUE(type,N)` will expand to a `typedef` definition with the name `Deque_typeN`.[15] For example,

40 ⟨`Deque` *example:* `Deque` *declarations* 40⟩≡

```
DECLARE_DEQUE(byte,3) ;
DECLARE_DEQUE(short,5) ;
```

will define the following `typedef` definitions:

41 ⟨`Deque` *example:* `Deque` *declarations expansions* 41⟩≡

```
typedef Deque_byte3 { ... } ;
typedef Deque_short5 { ... } ;
```

[14] In the programming language C++ we would have made `i` a `static` variable to the class `Deque`.

[15] The reader familiar with C++ [21] will recognise the similarity between the `DECLARE_DEQUE` construct and C++'s template mechanism. In fact, early C++ compilers used macro expansion to implement templates.

`PUSH_FRONT(deq,x)`	adds a new element `x` at the front of `deq`
`PUSH_BACK(deq,x)`	adds a new element `x` at the back of `deq`
`FRONT(deq)`	returns the first element of `deq`
`BACK(deq)`	returns the last element of `deq`
`POP_FRONT(deq)`	removes the first element of `deq`
`POP_BACK(deq)`	removes the last element of `deq`
`CLEAR(deq)`	removes all elements from `deq`
`COPY(src,dest)`	copies the elements of `Deque src` to `Deque dest`
`PRINT(deq)`	writes the contents of `deq` to the standard output
`SIZE(deq)`	returns the number of elements in `deq`
`IS_EMPTY(deq)`	returns `true` if `deq` does not contain any elements

Table 9. `Deque` operations.

To ease the declaration of `Deque` variables, we also define a short hand for the `Deque_...` names:

42 ⟨`Deque` *definition* 39⟩+≡ (48) ◁39

```
#define DEQUE(T,MAX) Deque_##T##MAX
```

Now we can introduce `Deque` variables as follows:

43 ⟨`Deque` *example:* `Deque` *variables* 43⟩≡

```
DEQUE(byte,3)    d1 ;
DEQUE(short,5)   d2 ;
```

`Deque` **operations** For the implementation of the `Deque` operations we use PROMELA's `inline` construct and `cpp` macros. We use PROMELA's `d_step` construct to encode the operations as efficient as possible. Table 9 shows the operations that we defined for `Deque` in [15].[16]

In this paper, we only present the definitions of `PUSH_FRONT`, `PUSH_BACK`, `COPY`, `FRONT`, `BACK`, `SIZE` and `IS_EMPTY`. The other `Deque` operations are left as an exercise.

44 ⟨`Deque` *operations* 44⟩≡ (48) 45▷

```
inline PUSH_FRONT(deq,x)
{
  d_step {
    deq.i=deq.length ;
    do
    ::  deq.i > 0  -> deq.a[deq.i]=deq.a[deq.i-1] ; deq.i--
    ::  deq.i == 0 -> break
    od ;
    deq.a[0]=x ;
    deq.length++ ;
    deq.i=0 ;
  }
}
```

[16] We borrowed the names for the `Deque` operations from C++'s Standard Template Library (STL) [21].

The operation `PUSH_FRONT(deq,x)` adds the element `x` to `deq` by shifting all elements of `deq` to the right in array `deq.a`. The 'local' field `deq.i` is used to iterate through the array `deq.a`. At the end of the operation this temporary variable is resetted to `0`.

The danger of `PUSH_FRONT` is that if the array `deq.a` is full, the operation will still try to add a new element. Fortunately, the `pan` verifier will trigger this "index out of bounds" error on run-time.[17] It would have nicer been though, if we had added an assertion like `assert(deq.length<N)` to the operation. Unfortunately, this is not possible as `N` is not fixed: there may be several `Deques` defined, all with different `MAX` arguments. We could have solved this by storing the size of the array into the `typedef` definition of `Deque`, but this would have enlarged the `Deque` objects.

`PUSH_FRONT` is an expensive operation: all elements in the `deque` have to be shifted one place to the right in order to insert a single element. Still, due to the `d_step` construct the complete operation only uses a single transition within SPIN.

The `Deque` type is a double-ended `List`, so we can also add elements to the back of the `Deque` object:

45 ⟨`Deque` *operations* 44⟩+≡ (48) ◁44 46▷

```
inline PUSH_BACK(deq,x)
{
  d_step {
    deq.a[deq.length]=x ;
    deq.length++ ;
  }
}
```

Like with `PUSH_FRONT`, there is no explicit check for an "index out of bounds" error. The operation `PUSH_BACK` is more efficient than `PUSH_FRONT`. In fact, when using an `array` to implement a `Deque` type, adding to the back of the `array` is always more efficient than to the front of the `array`.[18]

Because `typedef` and `array` objects in PROMELA are not assignable, we also need a operation to copy the contents of one `Deque` variable to another `Deque` variable.

46 ⟨`Deque` *operations* 44⟩+≡ (48) ◁45 47▷

```
inline COPY(src,dst)
{
  d_step {
    CLEAR(dst) ;
    dst.length=src.length ;
    dst.i=0 ;
    do
```

[17] Unless the `pan` verifier has been compiled using the directive `-DNOBOUNDCHECK`.

[18] It will be clear that the implementation of a `Stack`-like datatype on top of an `array` is most efficient: addition and removal of elements is always done at the back of the array.

```
      ::  dst.i <  dst.length -> dst.a[dst.i] = src.a[dst.i] ; dst.i++
      ::  dst.i >= dst.length -> break
      od ;
      dst.i=0 ;
    }
  }
```

Note that we first call `CLEAR` on `dst` to set all elements of `dst.a` to 0.
Below we define the operations on `Deque` objects that return a value:

47 ⟨`Deque` *operations* 44⟩+≡ (48) ◁46

```
#define FRONT(deq)      (deq.a[0])
#define BACK(deq)       (deq.a[deq.length-1])
#define SIZE(deq)       (deq.length)
#define IS_EMPTY(deq)   (deq.length==0)
```

Note that the macros `FRONT` and `BACK` do not check whether `deq` is non-empty.

Here our `Deque` implementation is ended. In [15] all `Deque` definitions and operations are defined and stored (using `noweb`) in a single 'header' file `deque.hpr`:

48 ⟨*deque.hpr* 48⟩≡

⟨`Deque` *definition* 39⟩
⟨`Deque` *operations* 44⟩

PROMELA models that need `Deque` objects can simply `#include` this file.

Conclusions In this recipe we have shown that PROMELA allows the definition of efficient user-defined types. A double-ended `Deque` type has been defined. In the same way, other abstract data types like single-ended `Queue`s, `List`s and `Stack`s can be defined and offered to the user via the usual `#include` mechanism.

Recipe 6 – Invariance

Manna and Pnueli [11] consider three main classes of temporal properties of reactive programs: *invariance*, *response* and *precedence* properties. This section is devoted to checking *invariance* properties with SPIN. An invariance property refers to a boolean expression P, and it requires that P is an invariant (i.e. is equal to `true`) over all reachable states of all computations [11]. In temporal logic notation, invariance properties are expressed by $\Box P$ for a state formula P.

Dwyer et. al. [4] have conducted a valuable survey on the practical use of temporal properties with respect to finite-state verification. They collected more than 500 temporal specifications to classify temporal properties into property patterns. One of the results of [4] is that 25% of the temporal properties that are being checked are invariance properties (i.e. universality or absence patterns in the terminoloy of [4]).[19]

For novice users of SPIN, the invariance property is easy to grasp and probably one of the first properties that they will verify with SPIN. There are several

[19] *Response* properties are even more common: they constitute nearly 50% of the temporal properties.

ways to verify an invariance property $\Box P$ with SPIN. In this recipe we discuss five of them. We have tested the different invariance schemes on several PROMELA specifications to find out which is most efficient. Our approach only allows references to *global* variables to appear in the expression P. This does not restrict the approach as local variables can always be declared globally. Using global variables may be less efficient than using local variables, though.

1. monitor process. The first method that we investigate is the method that is proposed in `assert.html` of [19]. This method is also the method of choice for people (relatively) new to SPIN. To express system invariance it suffices to place the invariant in an independently executed process.

49 ⟨*invariance - monitor process* 49⟩≡
```
active proctype monitor()
{
  assert(P) ;
}
```

Since the `monitor` process is executed independently from the rest of the system, the `assert(P)` statement may be evaluated at any time. Alternatively, one could add the `assert` statement to the `init` process after all processes have been started. Note that in this case the property P is not checked in the initial state of the system.

Even before running experiments with SPIN, however, we can predict that the 'independence' execution of the `monitor` process will be expensive. As the `assert` statement will be enabled in all states of the system, the number of states could – in the worst case – be doubled.

2. `never` claim - `do assert`. The SPIN documentation [19] also suggests another method to check for invariance.

50 ⟨*invariance* - `never do assert` 50⟩≡
```
never {
  do
  :: assert(P)
  od
}
```

The `never` claim ensures that after every step of the system the assertion is checked. In this way the number of states is not doubled, only the search depth of the verification run.

A minor drawback of this method is the fact that SPIN always gives the following warning after verifying a `never` claim:

```
warning: for p.o. reduction to be valid the never claim must be stutter-closed
(never claims generated from LTL formulae are stutter-closed)
```

As this `never` claim is not generated from a LTL formula, the novice SPIN user is not likely to trust the verification results after this warning about 'stutter-closed'-ness. In [15], stuttering is discussed in more detail. Here we only assure the reader that the `never do assert` method is always safe with respect to partial order reduction. And the warning can thus be ignored.

3. LTL property. The most *logical* way to check for invariance is to use SPIN's support for Linear Temporal Logic (LTL) formulae. SPIN's command line option `-f` translates a LTL formula to a `never` claim, encoding the corresponding Büchi acceptance condition.
The LTL formula $\Box P$ is translated to the following (stutter-closed) `never` claim:

51 ⟨*invariance* - *LTL* `never` *claim* 51⟩≡
```
never {
T0_init:
  if
  ::  (!P) -> goto accept_all
  ::  (1)  -> goto T0_init
  fi ;
accept_all:
  skip
}
```

4. guarded monitor process. A drawback of the ⟨*invariance* - *monitor process* 49⟩ method is that the `assert` statement is enabled in *every* state. To verify $\Box P$, though, it suffices to check that $\Diamond\neg P$ does not hold.

52 ⟨*invariance* - *guarded monitor process* 52⟩≡
```
proctype monitor()
{
end:
  atomic { !P -> assert(P) ; }
}
```

The `atomic` statement only becomes executable when `P` itself is not `true`. The `end` label is needed because if the `atomic` clause never becomes executable, the `monitor` process would have a 'non-valid end-state'.

5. `unless`. Our last method to check for invariance uses PROMELA's `unless` statement. The idea is to enclose the ⟨*body*⟩ of one of the processes of the system into the following `unless` clause:

53 ⟨*invariance:* `unless` 53⟩≡
```
proctype Foo()
{
  { ⟨body⟩
  } unless { atomic { !P -> assert(P) ; } }
}
```

Whenever `P` becomes `false`, the ⟨*body*⟩ will be interrupted and SPIN will conclude that the invariant property P does not hold. The `unless` method has some advantages, but these are outweigthed by the disadvantages:

+ No extra `proctype` is needed, which saves 4 bytes in the state vector.
+ The local variables of the process can also be used in the property P.
− The definition of a `proctype` has to be changed. This involves even more work when the process contains labels and `goto`'s.
− The `unless` construct can reach inside `atomic` clauses, which means that if the property P is `false` inside an `atomic` clause, the `unless` method will erroneously report an error.
− The partial order reduction may be invalid if rendez-vous communication is used within the body.
− The ⟨*body*⟩ of the process is not allowed to end, because otherwise the `unless` statement also terminates, preventing subsequent tests on `!P`.

6.1 Comparison

To compare the different invariance methods we used these methods to verify the following four (standard) PROMELA specifications:

brp	a bounded retransmission protocol (from [3]).
leader	a leader election protocol (part of SPIN's 3.3.10 distribution).
philo	a model for the well-known dining philosophers problem; we used `N=7` for the `default` runs and `N=6` for the `-DNOREDUCE` runs.
pftp	a flow control protocol (from [6] and part of SPIN's 3.3.10 distribution).

We conducted two types of verification runs. In the `default` case, we used SPIN's default settings and only adjusted the depth of the depth first search (DFS) stack (via option `-m`) when needed. Because the `unless` method is not reliable in combination with rendez-vous communication and partial order reduction (i.e. for the brp and philo runs), we repeated the verification runs with partial order reduction disabled (i.e, `-DNOREDUCE`).

Tables 10-13 list the results of verifying some trivial invariance properties using the methods discussed. The columns correspond with the `5` invariance methods. We verified two versions of the 'monitor process' method: in `1a` the `monitor` process is started first, whereas in `1b` the `monitor` process is started last. The 'best' results in a row are typeset using **boldface**.

Tables 10 and 12 report the total memory used by the verification runs in the `default` and `-DNOREDUCE` case, respectively. Tables 11 and 13 report the verification time (i.e. user+system time) of the runs. Due to space considerations, we have not included other significant parameters of the verification runs, like 'depth reached' or 'number of states stored'.

Conclusions First we consider the results in the columns `1a` and `1b`. The only difference between the PROMELA models of `1a` and `1b` is the activation order of the `monitor` process. Still the results for `1a` and `1b` show significant differences. If the `monitor` process is started last (`1b`) the verification statistics are worse. The reason for this is that SPIN's DFS will select the processes in reverse order; i.e. the last process started will be considered first. And because the `monitor`

	1a	1b	2	3	4	5
brp	27.157	38.115	**14.971**	15.688	26.338	22.135
leader	**2.542**	**2.542**	**2.542**	**2.542**	**2.542**	9.648
philo	16.318	21.336	**11.710**	11.915	**11.710**	12.510
pftp	**9.441**	11.285	**9.441**	9.953	**9.441**	9.744

Table 10. Invariance `default`: total memory used (Mb).

	1a	1b	2	3	4	5
brp	11.11	16.42	**4.90**	5.77	9.39	10.24
leader	0.03	**0.02**	**0.02**	0.03	**0.02**	3.36
philo	23.26	35.11	11.41	13.84	**11.31**	12.01
pftp	**1.53**	2.01	1.67	2.06	1.56	1.62

Table 11. Invariance `default`: verification time (sec).

	1a	1b	2	3	4	5
brp	40.777	59.415	22.140	24.735	40.572	**21.048**
leader	7.725	10.387	**5.062**	5.307	**5.062**	14.011
philo	11.504	16.317	**7.203**	7.835	**7.203**	17.973
pftp	30.644	40.679	**20.608**	22.037	**20.608**	22.307

Table 12. Invariance `-DNOREDUCE`: total memory used (Mb).

	1a	1b	2	3	4	5
brp	24.13	37.16	10.86	12.54	21.09	**9.60**
leader	2.66	4.18	1.30	1.37	**1.19**	5.76
philo	19.67	30.67	9.61	12.07	**9.57**	35.27
pftp	13.70	19.34	8.19	9.22	**7.84**	8.34

Table 13. Invariance `-DNOREDUCE`: verification time (sec).

process is always enabled, this step will always be executed before any other process can advance a step. But although `1a` performs better than `1b`, its results are still worse than the other invariance methods. So we conclude that to check for invariance one should not use the 'monitor process' solution. But if you do, be sure to activate the `monitor` process as the first process.

Although the `unless` method sometimes shows the best statistics (i.e. when partial order reduction is disabled) it has too many restrictions to be general applicable. ¿From the other three methods, method '4. guarded monitor process' seems to perform quite well. However, the results of method '2. `never do assert`' are always close or better. For the verification of the brp, method 2 performs even much better than method 4. It is also interesting to see that, although it never performs bad, there is no verification run where method '3. LTL property' shows the best results. Using this method to check for invariance is not a bad choice, but method 2 and 4 perform better.
To conclude we recommend to use method '2. `never do assert`' when checking invariance with SPIN.

Conclusions

In this paper we presented six 'recipes' to cook more efficient PROMELA models and to use the model checker SPIN more effectively. In Recipe 1 we showed how macros and `inline`s can help to structure and parameterise PROMELA models. Recipe 2 discussed some issues regarding the `atomic` and `d_step` constructs. In Recipe 3 we investigated the most efficient way to model randomness in PROMELA. In Recipe 4 we developed a `bitvector` library that is more efficient than SPIN's own `byte`-array implementation. In Recipe 5 we combined the ingredients of Recipe 1 and 2 to show how to add efficient data types to PROMELA. And in Recipe 6 we investigated the effectiveness of five different methods to check for invariance with SPIN.

This paper and the forthcoming [15] are not claimed to constitute a complete and finished collection of best practices for SPIN. On the contrary, the author hopes that this collection of techniques will stimulate other SPIN users to contribute their own best practices and experiences to this list. In this way not only the common knowledge on modelling and verification with SPIN will grow, it will also yield opportunities to optimise and improve the SPIN system itself.

Acknowledgements First of all, the author wants to thank Gerard Holzmann, the SPIN master, who is always available for patiently answering naive questions and personal wishes regarding SPIN. I would like to thank Pim Kars for showing me – already in 1996 – several efficient PROMELA tricks, that changed my attitude towards the application of verification tools. Ed Brinksma is thanked for sharing his experiences as a 'novice' SPIN user (i.e. Recipe 2 and 4). Yaroslav Usenko is thanked for his suggestion for a more elegant and slightly more efficient `do` solution for the `random` construct (i.e. Recipe 3). I want to thank Rom Langerak and especially the anonymous referees for their very useful suggestions to improve both the contents and readability of this paper.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.
2. Dov Bulka and David Mayhew. *Efficient C++ (Performance Programming Techniques).* Addison-Wesley, Reading, Massachusetts, 2000.
3. Pedro R. D'Argenio, Joost-Pieter Katoen, Theo C. Ruys, and G. Jan Tretmans. The Bounded Retransmission Protocol must be on time! (Full Version). CTIT Technical Report Series 97-03, Centre for Telematics and Information Technology, University of Twente, Enschede, The Netherlands, 1997. Also available from URL: http://wwwtios.cs.utwente.nl/~dargenio/brp/.
4. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 411–420, Los Angeles, CA, U.S.A., May 1999. ACM Press.
5. Gerard J. Holzmann. SPIN homepage: http://netlib.bell-labs.com/netlib/spin/whatispin.html.

6. Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.
7. Gerard J. Holzmann. SPIN Model Checking - Reliable Design of Concurrent Software. *Dr. Dobb's Journal*, pages 92–97, October 1997.
8. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
9. Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice Hall, second edition, 1988.
10. Donald E. Knuth. *Literate Programming.* Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information (CSLI), Stanford University, California, 1992.
11. Zohar Manna and Amir Pnueli. Tools and Rules for the Practicing Verifier. In R.F. Rashid, editor, *Carnegie Mellon Computer Science: A 25th Anniversary Commemorative*, pages 125–159. ACM Press, New York, 1991.
12. Scott Meyers. *Effective C++ (50 Specific Ways to Improve Your Programs and Designs).* Addison-Wesley, Reading, Massachusetts, second edition, 1998.
13. Norman Ramsey. `noweb` – homepage. Available from URL: http://www.cs.virginia.edu/~nr/noweb/.
14. Norman Ramsey. Literate Programming Simplified. *IEEE Software*, 11(5):97–105, September 1994.
15. Theo C. Ruys. Effective SPIN. CTIT Technical Report Series, Centre for Telematics and Information Technology, University of Twente, Faculty of Computer Science, Enschede, The Netherlands, August 2000. To appear.
16. René Seindal. *GNU m4, version 1.4.* Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA, 1.4 edition, November 1994. Available from URL: http://www.gnu.org.
17. SPIN Documentation. Basic SPIN Manual. Part of SPIN's online HTML documentation.
18. SPIN Documentation. Proceedings of the SPIN Workshops. Part of SPIN's online HTML documentation.
19. SPIN Documentation. SPIN Version 3.3: Language Reference - Man-Pages and Semantics Definition. Part of SPIN's online HTML documentation.
20. SPIN Documentation. What's New in SPIN Versions 2.0 and 3.0 - Summary of changes since Version 1.0. Part of SPIN's online HTML documentation.
21. Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, third edition, 1997.