# The High Road to Formal Validation:

## Model Checking High-Level versus Low-Level Specifications

Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
leuschel@cs.uni-duesseldorf.de

**Abstract.** In this paper we examine the difference between model checking high-level and low-level models. In particular, we compare the ProB model checker for the B-method and the Spin model checker for Promela. While Spin has a dramatically more efficient model checking engine, we show that in practice the performance can be disappointing compared to model checking high-level specifications with ProB. We investigate the reasons for this behaviour, examining expressivity, granularity and Spin's search algorithms. We also show that certain types of information (such as symmetry) can be more easily inferred and exploited in high-level models, leading to a considerable reduction in model checking time.
**Keywords:** B-Method, Tool Support, Model Checking, Symmetry Reduction, Spin.[1]

## 1 Introduction

Model checking [11] is a technique for validating hardware and software systems based on exhaustively exploring the state space of the system. Model checking has been hugely successful and influential, culminating in the award of the Turing Prize to Clarke, Emerson and Sifakis.

Most model checking tools work on relatively low-level formalisms. E.g., the model checker smv [29, 9] works on a description language well suited for specifying hardware systems. The model checker Spin [19, 21, 5] accepts the Promela specification language, whose syntax and datatypes have been influence by the programming language C. Recently, however, there have also been model checkers which work on higher-level formalisms, such as ProB [24, 27] which accepts B [1]. Other tools working on high-level formalisms are, for example, fdr [16] for CSP and alloy [23] for a formalism of the same name (although they both are strictly speaking not model checkers).

It is relatively clear that a higher level specification formalism enables a more convenient modelling. On the other hand, conventional wisdom would dictate

[1] This research is being carried out as part of the DFG funded research project GEPAVAS and the EU funded FP7 research project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

that a lower-level formalism will lead to more efficient model checking. In this paper we try to establish that this is not necessarily the case; sometimes it can even be considerably more efficient to directly validate high-level models.

In this paper we concentrate on comparing two formalisms: the high-level B-method and the more low-level specification language Promela, as well as the associated model checkers ProB and Spin.

Indeed, Spin is one of the most widely used model checkers. Spin is an impressive feat of engineering and has received the ACM System Software Award in 2001. A lot of research papers and tools [7, 31, 3, 10, 18, 35, 32] translate other formalisms down to Promela and use Spin. Against Spin we pit the model checker ProB, which directly works on a high-level specification language,[2] but has a much less tuned model checking engine.

In Section 2 we first examine the granularity and expressivity of both formalisms, and explain what can go wrong when translating a high-level formalism down to Promela. In Section 3 we examine the problems that can arise when using Spin on systems with a very large state space (such as typically encountered when model checking software). In Section 4, we look at one piece of information (symmetry) that can be more easily extracted from high-level models, and its impact on the performance of model checking.

## 2 Granularity and Expressivity

B being at a much higher-level than Promela, it is clear that a single B expression can sometimes require a convoluted translation into Promela. As an example, take the following B operation to sort an array `a` of integers:

```
Sort = ANY p WHERE p:perm(dom(a)) &
                   !(i,j).(i:1..(size(a)-1) & j:2..size(a) & i<j
                            =>  a(p(i)) <= a(p(j))) THEN
          a := (p;a) END;
```

The sorting operation is specified as choosing any permutation `p` such that after applying the permutation the array is sorted. Note that `(p ; a)` corresponds to relational composition, and `(p;a)(i) = a(p(i))`. Promela does not have such a powerful non-deterministic construct. Basically, sorting will have to be encoded in Promela by actually implementing a sorting algorithm, which will take up considerably more space and time to write than the above B specification (see, e.g., the merge sort example accompanying the book [4]).

But even less extreme examples are sometimes surprisingly difficult to model in Promela. Take, e.g., the B statement `x::1..n`, which non-deterministically sets `x` to a value between 1 and $n$. In Promela, we have to encode this using an if-statement as follows:

[2] See [2] which argues that formal verification tools should tie directly to high-level design languages.

```
if
  :: x=1
  :: x=2
  (...)
  :: x=n
fi
```

This translation is not very concise, and moreover can only be performed if we statically know the value of $n$. In case $n$ is a variable, we have to encode `x::1..n` in Promela as follows (see, Section 4.6.2 in [5]):

```
x = 1;
do
  :: (x<nn) -> x++
  :: ((x>=1)&&(x<=nn)) -> break
od
```

This translation is more concise (for larger $n$) than using an if-statement, but still adds unnecessary state and behaviours to the model. Figure 1 shows on the left the behaviour of the B model, starting from a state where the variable `x` has the value 2 and supposing that $n = 4$. On the right we see the behaviour of our Promela translation, which has $n = 4$ additional intermediate states.

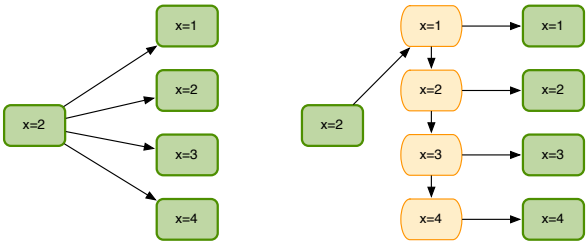


**Fig. 1.** B and Promela state space for non-deterministic choice `x::1..n` with $n = 4$

In [21] on page 462 another translation is suggested, which produces a more random distribution when animating. However, this translation has the disadvantage that the `do` loop may not terminate (which can cause a problem with verification; see below).

The situation depicted in Figure 1 is actually quite typical: by translating a specification from B to Promela we add new intermediate states and additional internal behaviour. As another example, take the B predicate `x:ran(f)`, where `f` is a total function.[3] In Promela, we could encode the function as an array,

[3] This is actually taken from our small case study in Section 4.2.

supposing for simplicity that the domain of the function is $0..n-1$ for some $n$. Evaluating the predicate `x:ran(f)` again requires a loop in Promela:

```
byte i = 0;
do
  :: (i<n && f[i]!=x) -> i++
  :: (i<n && f[i]==x) -> break    /* not(x:ran(f)) */
  :: (i==n) -> break              /* x:ran(f) */
od;
```

Again, additional intermediate states appear in the Promela model. Note, that in the presence of concurrency, these additional states can quickly lead to a blow-up in the number of states compared to the high-level model. E.g., if we run four copies of the B machines from Figure 1 in parallel, we get $5^4 = 625$ states. If we run four copies of the Promela specification from Figure 1 in parallel, we get $9^4 = 6561$ states.

A similar picture arises for the `x:ran(f)` example. Supposing we have $n = 7$ and that we have 4 processes, the B model would have, for a given function $f$, $2^4 = 16$ states (before and after evaluating the predicate). The Promela model will have in the worst case (where the predicate `x:ran(f)` is false for all processes) $10^4 = 10,000$ states. Also, we have the problem that the new variable $i$ is now part of the state. In the Promela model we should ensure that $i$ is reset to some default value after being used; otherwise a further blow-up of the state space will ensue.

Luckily, Promela provides the `atomic` construct, which can be used to avoid the additional blow-up due to internal states. However, the `atomic` construct does have some restrictions (e.g., in the presence of channel communications). Also, care has to be taken when placing an `atomic` around `do` loops: if the loop does not necessarily terminate, then exhaustive verification with SPIN will become impossible (the "search depth too small" error will be systematically generated). If by accident the loop does not terminate at all, as in the following example, then SPIN itself will go into an infinite loop, without generating an error message or warning:

```
atomic{ do
            :: x<10 -> x++
            :: x>0 -> x--
        od }
```

In summary, translating high-level models into Promela is often far from trivial. Additional intermediate states and additional state variables are sometimes unavoidable. Great care has to be taken to make use of `atomic` (or even `dstep`) and resetting dead temporary variables to default values. However, restrictions of `atomic` make it sometimes very difficult to hide all of the intermediate states.

As far as an automatic translation is concerned, it is far from clear how B could be automatically translated into Promela;[4] it is actually already very challenging to write an interpreter in Prolog for B [24, 27].

**A small empirical study**

[36] studies the elaboration of B-models for ProB and Promela models for SPIN on ten different problems. With one exception (the Needham-Schroeder public key protocol), all B-models are markedly more compact than the corresponding Promela models. On average, the Promela models are 1.85 longer (counting the number of symbols). The time required to develop the Promela models was about 2-3 times higher than for the B models, and up to 18 times higher in extreme cases. No model took less time in Promela. Some models could not be fully completed in Promela. The study also found that in practice both model checkers PROB and SPIN were comparable in model checking performance, despite PROB working on a much higher-level input language and being much slower when looking purely at the number of states that can be stored and processed. In the remainder of this paper, we will investigate the possible reasons for this surprising fact, which we have ourselves witnessed on repeated occasions, e.g., when teaching courses on model checking or conducting case studies.

## 3 Searching for Errors in Large State Spaces

In this section we will look at a simple problem, with simple datatypes, which can be easily translated from B to Promela, so that we have a one-to-one correspondence of the states of the models. In such a setting, it is obvious to assume that the SPIN model checker for Promela will outperform the B model checker by several orders of magnitude. Indeed, SPIN generates a specialised model checker in C which is then compiled, whereas PROB uses an interpreter written in Prolog. Furthermore, SPIN has accrued many optimisations over the years, such as partial order reduction [22, 30] and bitstate hashing [20]. (PROB on its side does have symmetry reduction; but we will return to this issue in the next section.)

The simple Promela example in Fig. 2 can be used to illustrate this speed difference. The example starts with an array in descending order, and then allows permutation at some position $i$, with $i$ cycling around the array. The goal is to reach a state where the array starts with `[1,2,3]`. On a MacBook Pro with a 2.33 GHz Core2 Duo, SPIN (version 4.2.9 along with XSpin 4.28) takes 0.00 seconds to find a solution for the Promela model (plus about six seconds compilation time on XSpin and 40 seconds to replay the counter example which is 1244 steps long). PROB (version 1.2.7) takes 138.45 seconds for the same task on an equivalent B model. This, however, includes the time to display the counter example, which is in addition also only 51 steps long. Still, the model checking speed is dramatically in favour of SPIN (and the difference increases further when using larger arrays).

[4] This process was actually attempted in the past — without success — within the EPSRC funded project ABCD at the University of Southampton.

```
#define SZ 8
active proctype flipper () {
    byte arr[SZ];  byte i,t;
    do
      :: i==SZ -> break
      :: i<SZ -> arr[i] = SZ-i; i++
    od;
    i = 0;
    do
      :: i<SZ-1 -> i++
      :: i==SZ-1 -> i=0
      :: i<SZ-1 -> t = arr[i]; arr[i]=arr[i+1]; arr[i+1]=t; t=0; i++
      :: (arr[0]==1 && arr[1]==2 && arr[2]==3)
          -> assert(false) /* found solution */
    od
}
```

**Fig. 2.** Flipping adjacent entries in an array in Promela

However, it is our experience that this potential speed advantage of SPIN often does not translate into better performance in practice in real-life scenarios. Indeed—contrary to what may be expected—we show in this section that SPIN sometimes fares quite badly when used as a debugging tool, rather than as verification tool. Especially for software systems, verification of infinite state systems cannot be done by model checking (without abstraction). Here, model checking is most useful as a debugging tool: trying to find errors in a very large state space.

### 3.1 An Experiment

Let us model a simple ticket vending machine, each ticket costing five euros. Those tickets can either be paid using a credit card or with coins. If no more tickets are available the machine should no longer accept coins or credit cards. Figure 3 depicts a low-level Promela specification of this problem. For simplicity, the machine requires the user to insert the exact amount (i.e., no change is given) and we have not yet specified a button which allows a user to recover inserted money. In the model we have also encoded that before issuing a ticket (via `ticket--`), the number of available tickets is greater or equal than 1.

Figure 3 actually contains an error: the credit card number (for simplicity always 1) is not reset after a ticket has been issued. This can lead to an assertion violation. An equivalent specification in the high-level B formalism is depicted in Figure 4. The same error is reproduced there, leading to an invariant violation.

Both models can also deadlock, namely when all tickets have been issued. Both problems have been fixed in adapted models. In the Promela version the `cardnr` variable is now reset to 0 after being used and the following line has been added as the last branch of the `do` loop:

```
    :: (ticket==0) -> ticket = 2 /* reload ticket machine */
```

Similarly, in the B model the `withdraw_ticket_from_card` has been corrected to reset the `cardnr` and the following operation has been added:

```
resupply = PRE ticket = 0 THEN ticket := 2 END;
```

Both models have also been enriched with an additional constraint, namely that not more than 70 coins should be inserted at any one time. The models do not yet ensure these constraints, hence the model checkers should again uncover assertion violations.

```
active proctype user () {
 byte c10 = 0; byte c20 = 0; byte c50 = 0;
 byte c100 = 0; byte c200 = 0; byte cardnr = 0;
 byte ticket = 2;
 do
    :: (cardnr==0 && ticket>0) -> c10++
    :: (cardnr==0 && ticket>0) -> c20++
    :: (cardnr==0 && ticket>0) -> c50++
    :: (cardnr==0 && ticket>0) -> c100++
    :: (cardnr==0 && ticket>0) -> c200++
    :: (c10+c20+c50+c100+c200==0 && ticket>0) -> cardnr = 1
    :: ((c10+2*c20+5*c50+10*c100+20*c200)==500)
          -> assert(ticket>0);
             atomic{ticket--; c10=0; c20=0; c50=0; c100=0; c200=0}
    :: (cardnr>0) ->  assert(ticket>0); ticket--
         /* forgot to reset cardnr */
 od
}
```

**Fig. 3.** A nasty ticket vending machine in Promela

We now compare using SPIN (version 4.2.9 along with XSpin 4.28) on the Promela model against using PROB (version 1.2.7) on the B model. All tests were again run on a MacBook Pro with a 2.33 GHz Core2 Duo.

The original Promela specification from Fig. 3 actually also had an additional, unintentional error: it contained `assert(ticket>=0)` instead of `assert(ticket>0)` for the last branch of the `do` loop. This surprisingly meant that SPIN could not find an assertion violation, as bytes are by definition always positive ($0-1=255$ for bytes in Promela).

After fixing this issue, a series of experiments were run using SPIN. The results can be found in Table 1. The model checking times are those displayed by SPIN (user time), and do not include the time to generate and compile the `pan` file (which takes about 6 seconds with XSpin). The very first line shows the use of SPIN on the model from Fig. 3, when used in default settings. As one can

```
MACHINE NastyTicketVending
DEFINITIONS SET_PREF_MAXINT == 255
VARIABLES
  c10,c20,c50,c100,c200, cardnr, ticket
INVARIANT
  c10:NAT & c20:NAT & c50:NAT & c100:NAT & c200:NAT & cardnr:NAT & ticket:NAT
INITIALISATION
  c10,c20,c50,c100,c200, cardnr, ticket := 0,0,0,0,0, 0,2
OPERATIONS
  insert_10cents = PRE cardnr=0 & ticket>0 THEN c10 := c10 + 1 END;
  insert_20cents = PRE cardnr=0 & ticket>0 THEN c20 := c20 + 1 END;
  insert_50cents = PRE cardnr=0 & ticket>0 THEN c50 := c50 + 1 END;
  insert_100cents = PRE cardnr=0 & ticket>0 THEN c100 := c100 + 1 END;
  insert_200cents = PRE cardnr=0 & ticket>0 THEN c200 := c200 + 1 END;
  insert_card = PRE c10+c20+c50+c100+c200=0 & ticket>1 THEN cardnr := 1 END;
  withdraw_ticket_from_coins = PRE c10+2*c20+5*c50+10*c100+20*c200=50  THEN
    c10,c20,c50,c100,c200, cardnr, ticket := 0,0,0,0,0, 0,ticket-1
  END;
  withdraw_ticket_from_card = PRE cardnr>0 THEN
     ticket := ticket -1  /* forgot to reset cardnr */
  END
END
```

**Fig. 4.** A nasty ticket vending machine in B

see, it took SPIN 40 seconds before aborting with an "out of memory" error. No counter-example was found. After that we successively adapted various of the (many) SPIN's parameters. It can be seen that even with bitstate hashing enabled, no error was detected. However, after turning on breadth-first search, an assertion violation was finally found.

We now turned to the second model, where the two problems of the first model were corrected. We started off with the setting that was successful for the first model; but this time this setting proved incapable of detecting the new error in the second model. Only after reverting back to a depth-first search was an assertion violation detected. (Note that it took XSpin several minutes to replay the counter example containing over 65000 steps.)

In summary, for the first deadlocking model (Fig. 3) it took us about 1000 seconds of CPU time and an afternoon to realise that the initial version of the model was wrong. After that it took us about 800 seconds (adding up the various runs of SPIN in the upper half of Table 1) of CPU time and 45 minutes in total to locate an error in the model.[5] For the equivalent high-level B specification

[5] This shows that one should be very careful about experimental results for a tool with many parameters: if only the successful runs get published (i.e., experiment 6 in Table 1 for the deadlocking model) the reader can get a very misleading picture of the real-life performance of a tool, as all the time and expertise required to tune the parameters is ignored.

| Search Depth | Memory MB | Partial Order | Bitstate Hashing | Breadth First | Time (sec) | Result |
|---|---|---|---|---|---|---|
| Deadlocking model from Fig. 3 | | | | | | |
| 10,000 | 128 | yes | no | no | 40.00 | out of memory † |
| 10,000 | 512 | yes | no | no | 580.26 | out of memory † |
| 10,000 | 512 | yes | yes | no | 89.59 | † |
| 100,000 | 512 | yes | yes | no | 91.51 | † |
| 1,000,000 | 512 | yes | yes | no | 97.04 | † |
| 100,000 | 512 | yes | yes | yes | 0.00 | error found |
| Non-deadlocking model with ticket resupply | | | | | | |
| 100,000 | 512 | yes | no | yes | 64.26 | out of memory |
| 100,000 | 512 | yes | yes | yes | 47.23 | out of memory |
| 100,000 | 512 | yes | yes | no | 0.17 | error found |

† = search depth too small

**Table 1.** SPIN experiments on the nasty vending machine

in Fig. 4, ProB took 0.2 seconds to find an invariant violation (with default settings). The counter-example consisted of 4 steps: one `insert_card`, followed by 3 `withdraw_ticket_from_card` events.[6]

For the non-deadlocking model, it took us in all about 111 seconds of CPU time and three attempts to uncover the error with SPIN. For the equivalent B model, ProB takes 24 seconds to find the invariant violation, again in default settings. Observe that the counter example consists of the minimally required 70 steps; SPIN's counter example consists of over 65000 steps.

### 3.2 Explanation of the Experimental Results

What can explain this poor performance of SPIN compared to ProB? The specification is quite simple, and the Promela and B models are very similar in size and complexity. On the technology side, SPIN compiles the Promela models to C and can crunch hundreds of thousands of states per second. ProB uses a Prolog interpreter to compute the state space of the B specification. SPIN uses partial order reduction, ProB does not (and symmetry does not apply here).

Let us first examine the characteristics of the models. The deadlocking model has a very large state space, where there is a systematic error in one of the operations of the model (as well as a deadlock when all tickets have been withdrawn). To detect the error, it is important to enable this operation and then exercise this operation repeatedly. It is not important to generate long traces of the system, but it is important to systematically execute combinations of the individual operations. This explains why depth-first behaves so badly on this model, as it will always try to exercise the first operation of the model first (i.e., inserting

[6] Depending on the run, a deadlock can also be found. We return to this later.

the 10 cents coin). Note that a very large state space is a typical situation in software verification (sometimes the state space is even infinite).

In the corrected non-deadlocking model the state space is again very large, but here the error occurs if the system runs long enough; it is not very critical in which order operations are performed, as long as the system is running long enough. This explains why for this model breadth-first was performing badly, as it was not generating traces of the system which were long enough to detect the error.

In order to detect both types of errors with a single model checking algorithm, ProB has been using a mixed depth-first and breadth-first search [27]. More precisely, at every step of the model checking, ProB randomly chooses between a depth-first and a breadth-first step. This behaviour is illustrated in Fig. 5, where three different possible runs of ProB are shown after exploring 5 nodes of the B model from Fig. 4.

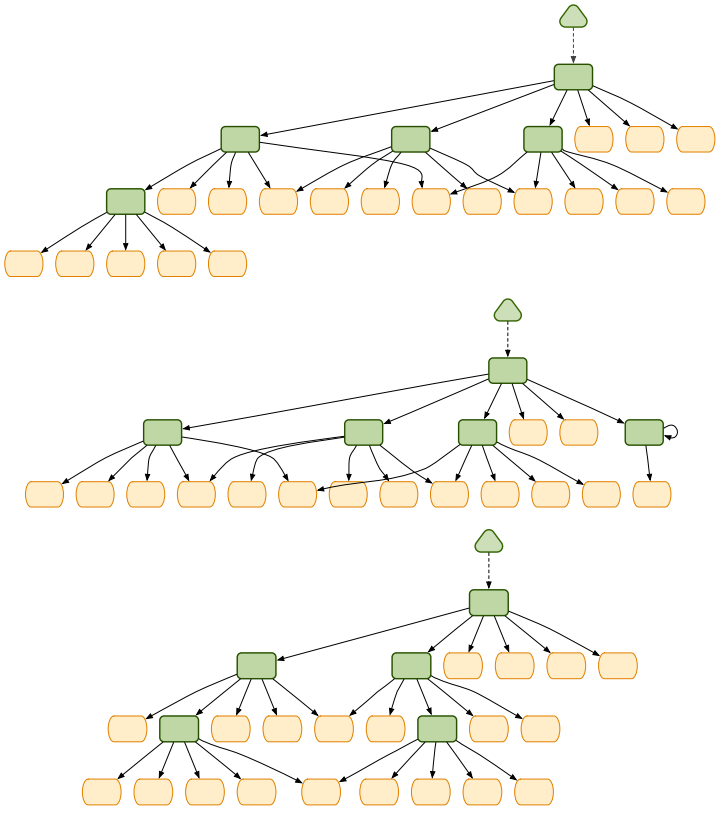

**Fig. 5.** Three different explorations of ProB after visiting 5 nodes of machine in Fig. 4

The motivation behind ProB's heuristic is that many errors in software models fall into one of the following two categories:

- Some errors are due to an error in a particular operation of the system; hence it makes sense to perform some breadth-first exploration to exercise all the available functionality. In the early development stages of a model, this kind of error is very common.
- Some errors happen when the system runs for a long time; here it is often not so important which path is chosen, as long as the system is running long enough. An example of such an error is when a system fails to recover resources which are no longer used, hence leading to a deadlock in the long run.

One may ask whether the random component of ProB's algorithm can lead to large fluctuations in the model checking time. Figure 6 shows the result of a small experiment, where we have timed 16 runs for the above deadlocking machine from Fig. 4. The average runtime was 0.46 seconds, the standard deviation was 0.36. As can be seen, in all cases an error was found reasonably quickly, the worst time being 1.31 seconds.

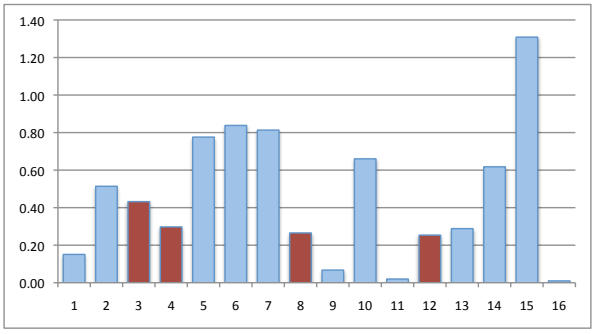


**Fig. 6.** 16 Runtimes for model checking Fig. 4; a deadlock was found in runs 3,4,8,12

In summary, if the state space is very large, Spin's depth-first search can perform very badly as it fails to systematically test combinations of the various operations of the system. Even partial order reduction and bitstate hashing do not help. Similarly, breadth-first can perform badly, failing to locate errors that require the system to run for very long. We have argued that ProB's combined depth-first breadth-first search with a random component does not have these pitfalls.

The aspect we have discussed in this section does not yet show a fundamental difference between model checking high-level and low-level models. Indeed, recently there has been considerable interest in directed model checking, using informed search strategies with heuristic functions such as A* or best-first-search,

see, e.g. [37] leading to [15, 14] in the context of Promela. However, for a low level formalism, one is probably much more reluctant to adopt those techniques as they may noticeably slow down the verification when not needed. (Indeed, the above mentioned techniques have not yet found their way into the official distribution of SPIN.) For high-level models, the overhead of adopting a more intelligent search algorithm is less pronounced, as processing individual states takes up considerably more time. Hence, there is scope for considerably more refined search algorithms when model checking high-level models.[7]

## 4 Exploiting Symmetry in High-level Models

In the previous section we encountered a scenario were complete verification was impossible (as the state space was too large), and model checking was used as a debugging tool. In this section we return to the verification scenario (although the points should also be valid for a debugging scenario), and show that even there it can be much more efficient to model check a high-level model than a low-level one. The reason is that in the high-level model certain properties, such as symmetries, can be detected much more easily than in a low-level model. Exploiting those properties in the model checker then leads to a considerably reduced state space.

For example in B, symmetries are induced by the use of deferred sets, as every element of such a set can be substituted for every other element [26]. The use of deferred sets is very common in B, and hence many B models exhibit symmetry which can be exploited by PROB [26, 34, 28, 33].

### 4.1 First Experiment: Scheduler

For a first experiment we have used the scheduler1.ref machine from [25] (also used in [26] and contained in Appendix B). Here `PROC` is a deferred set (of process identities) and the translation can be found in Appendix A. Comparing different tools and formalisms is always a difficult task. We have obtained the help of Promela/SPIN experts to construct a faithful Promela counterpart of the B model. We have taken extreme care to translate this B specification into the best Promela code possible (with the help of Alice Miller and Alastair Donaldson) and have ensured that both models exhibit the same number of states (in the absence of symmetry reduction and partial order reduction). Also, the example is not artificially chosen or constructed so as to make our tool behave better. It was chosen because it was a relatively simple B model, that could still be hand translated with reasonable effort into an equivalent Promela model (it also has the property that symmetry in the B model can be translated into symmetry of process identifiers in the Promela model; something which will no longer be true for more complicated models, e.g., from [28] or the model we present later in Section 4.2).

[7] Within the DFG-funded project GEPAVAS we are actually investigating adding heuristics when model checking B specifications.

Still, this is just one example and we do not claim that the phenomena uncovered here is universal. Indeed, as we have already seen in Section 3, if one takes a B model with no deferred sets (and hence no symmetry exploitable by ProB), then complete verification with Spin will be substantially faster than our tool (provided the B model can be translated into Promela). But our intuition is that, for some application domains, working at a higher level of abstraction (B vs. Promela) can be beneficial both for the modelling experience of the user (less effort to get the model right) and for the model checking effort.

In the experiments below, we have used the same setup as before in Section 3. This time we have incorporated the ProB and Spin results in the single Table 2. To exploit the symmetry in ProB we have used the technique from [33] based on the nauty graph canonicalisation package. In addition to timings reported by the two tools, we have also used a stopwatch to measure the user experience (these timings were rounded to the nearest half second). For Spin, default settings were used, except where indicated by the following symbols: $c1$ means compression (c1) was used, $\odot$ meaning bitstate hashing (DBITSTATE) was used, $1GB$ means that the allocated memory was increased to 1 GB of RAM, $>$ signifies that the search depth was increased to 100,000, $\gg$ that the search depth was increased to 1,000,000, and $\ggg$ that search depth was increased to 10,000,000. The ProB time includes time to check the invariant. Only deadlock and invalid end state checking is performed in Spin.

| Card | Tool | States | Time | Stopwatch |
|---|---|---|---|---|
| 2 | ProB | 17 | 0.04 s | $<$ 0.5 s |
| | ProB + nauty | 10 | 0.03 s | $<$ 0.5 s |
| | Spin (default) | 17 | 0.02 s | 6 s |
| 4 | ProB | 321 | 1.08 s | 1.5 s |
| | ProB+ nauty | 26 | 0.15 s | $<$ 0.5 s |
| | Spin (default) | 321 | 0.00 s | 6 s |
| 8 | ProB+ nauty | 82 | 1.18 s | 1.5 s |
| | Spin (default) | † 483980 | 2.50 s | 6.5 s |
| | Spin ($> c1$) | † 545369 | 5.84 s | 11 s |
| | Spin ($\gg$) | 595457 | 3.75 s | 6 s |
| | Spin ($\gg c1$) | 595457 | 7.47 s | 11 s |
| 12 | ProB+ nauty | 170 | 4.90 s | 5.5 s |
| | Spin ($\gg c1$) | †1.7847e+06 | 17.92 s | 22 s |
| | Spin ($\ggg c1$ 1GB ) | $\propto$ † 1.1877e+07 | 135.60 s | 140 s |
| | Spin ($\ggg c1$ 1GB $\odot$) | † 4.0181e+07 | 295.71 s | 302 s |

† = search depth too small, $\propto$ = out of memory

**Table 2.** Experimental results for scheduler1

One can observe that for 2 and 4 processes ProB with symmetry reduction is actually quite competitive compared to Spin with partial order reduction, despite the much higher-level input language. Furthermore, if we look at the total time taken to display the result to the user measured with a stopwatch,

ProB is faster (for Spin there is the overhead to generate and compile the C code). For 8 processes, ProB is about three times faster than Spin. Note that it took us considerable time to adjust the settings until Spin was able to fully check the model for 8 processes.[8] For 12 processes we were unable to exhaustively check the model with Spin, even when enabling bitstate hashing. Attempts to further increase the search depth led to "command terminated abnormally." ProB checked the model for 12 processes in 5.5 seconds.

Of course, in addition to partial order reduction, one could also try and use symmetry reduction for Spin, e.g., by using the SymmSpin tool [6] or TopSpin tool [13]. To use TopSpin a minor change to the Promela model is required, after which the model with 8 processes can be verified in 0.09 s with combined partial order reduction and symmetry (not counting the compilation overhead). However, compared to ProB's approach to symmetry, we can make the following observations:

1. In Promela the user has to declare the symmetry: if he or she makes a mistake the verification procedure will be unsound; (there is, however, the work [12] which automatically detects some structural symmetries in Promela). In B symmetries can be inferred automatically very easily.
2. Symmetry is much more natural and prevalent in B and ProB can take advantage of partial symmetries (see, e.g., the generic dining philosophers example in [28]) and one can have multiple symmetric types (for Spin typically only a single scalarset, namely the process identifiers, is supported). In TopSpin all processes must be started in an atomic block; in B constants can be used to assign different tasks or behaviours to different deferred set elements; the partial symmetries can still be exploited.
3. Using and installing the symmetry packages for Spin is not always straightforward (the packages patch the C output of Spin). TopSpin cannot as of now be downloaded.

To further illustrate point 2, we now show a model with multiple deferred sets. To the best of our knowledge, this model cannot be put into a form so that TopSpin can exploit the symmetries.

### 4.2 Second Experiment: Multiple Symmetric Datatypes

Figure 7 contains a small B model of a server farm, with two deferred sets modelling the users and the servers. Individual servers can connect and disconnect from the system, and the system routes user requests to an available server via the `UserRequest` operation, trying to maintain the same server for the same user on later occasions (unless a timeout occurs). The time to develop and model check the model with ProB was about 10 minutes.

[8] The development of the model itself also took considerable time (and several email exchanges with Alice Miller and Alastair Donaldson); the first versions exhibited much worse performance when used with Spin.

Figure 8 contains a Promela version of the same problem. The Promela model was actually simplified: the second do loop always takes the first available server rather than non-deterministically choosing one.[9]

It took about one hour and a half until we had a model which could be checked for cardinality 6 (local variables had to be reset to 0; errors had crept up in the loops to find server machines, etc.). In Table 3 we show the results of our experiments with SPIN. Observe that for cardinality of 8 we did not manage to verify the model using SPIN. Complete model checking with PROB for the same cardinality takes 3.48 seconds with nauty and 0.77 seconds with the hash marker method from [28]. For a cardinality of 9, PROB takes 21.04 seconds with nauty and 1.16 seconds with the hash marker method.

It may be possible to further improve the Promela model (but note that we have already put about 10 times the effort into the Promela model than into the B model). In summary, the symmetry that can be inferred in the high-level model again leads to a dramatic reduction in model checking time.

```
MACHINE ServerFarm
SETS USERS;SERVER
VARIABLES active, serving
INVARIANT active <: SERVER & serving: active >+> USERS
INITIALISATION active,serving := {},{}
OPERATIONS
  ConnectServer(s) = PRE s:SERVER & s/: active THEN
      active := active \/ {s} END;
  DisconnectServer(s) = PRE s:SERVER & s:active THEN
    active := active - {s} || serving := {s} <<| serving END;
  s <-- UserRequest(u) =  PRE u:USERS THEN
    IF u:ran(serving) THEN
       s := serving~(u)
    ELSE
       ANY us WHERE us:SERVER & us : active & us /: dom(serving) THEN
           s:= us || serving(us) := u END
    END
  END;
  UserTimeout(u) = PRE u:USERS & u:ran(serving) THEN
     serving := serving |>> {u} END
END
```

**Fig. 7.** A server farm model in B

[9] One solution would be not to force the loop to chose the first available server. However, to avoid deadlocks, one should then also allow decrementing `i`. But then Spin will never be able to exhaustively check the model, unless we remove the `atomic` surrounding the loop. I.e., the proper solution would be to adapt the data structure, e.g., remembering also how many servers are still available.

```
chan connect = [0] of { byte } ;
chan disconnect = [0] of { byte } ;
chan request = [0] of { byte } ;
#define SERVERS 8
#define USERS 8

active[SERVERS] proctype server () { /* pids from 0.. SERVERS-1 */
   do
     :: connect!_pid -> disconnect!_pid
   od
}
active[USERS] proctype user () { /* pids start at SERVERS */
   do
     :: request!_pid
   od
}
active proctype mserver () {
    bit sactive[SERVERS];
    byte serving[SERVERS];
    byte x = 0;
    do
      :: atomic{connect?x -> assert(sactive[x]==0) ->
          assert(serving[x]==0) -> sactive[x]=1 ->x=0}
      :: atomic{disconnect?x -> assert(sactive[x]==1) ->
          sactive[x]=0 -> serving[x]=0-> x=0}
      :: atomic{request?x;
        byte i = 0;
        do
          :: (i<SERVERS && serving[i]!=x) -> i++
          :: (i<SERVERS && serving[i]==x) -> break
          :: (i==SERVERS) -> break
        od;
        if
          :: (i==SERVERS) -> i=0 -> do
             :: (i<SERVERS && (sactive[i]==0 || serving[i]!=0)) -> i++
             :: (i<SERVERS && sactive[i]!=0 & serving[i]==0)
                    -> serving[i] = x -> break
             :: (i==SERVERS) -> printf("no server available") -> break
            od;
          :: (i<SERVERS) -> printf("already connected")
        fi;
        x = 0;i=0 /* reset x,i to avoid state explosion */
        }
    od
 }
```

**Fig. 8.** A server farm model in Promela

| Card | Search Depth | Memory MB | Partial Order | Bitstate Hashing | Breadth First | Time (sec) | Result |
|---|---|---|---|---|---|---|---|
| 6 | 100,000 | 512 | yes | no | no | 1.74 | † |
| | 1,000,000 | 512 | yes | no | no | 2.26 | ok |
| 7 | 1,000,000 | 512 | yes | no | no | 22.71 | † |
| | 10,000,000 | 512 | yes | no | no | 32.21 | ok |
| 8 | 10,000,000 | 512 | yes | no | no | 82.05 | † |
| | 100,000,000 | 512 | yes | no | no | - | error |
| | 10,000,000 | 512 | yes | yes | no | 279.37 | † |
| | 100,000,000 | 512 | yes | yes | no | - | error |

† = search depth too small; error = "command terminated abnormally"

**Table 3.** SPIN on the server farm from Fig. 8

## 5 Conclusion

SPIN is an extremely useful and very efficient model checking tool. Still, over the years, we have accumulated a certain amount of anecdotal evidence which shows that using a model checker for high-level models can quite often give much better results in practice. In this paper we have investigated the reasons for this counterintuitive behaviour.

In Section 2 we have studied the granularity and expressivity of Promela versus B, and have shown that the Promela counterpart of a B model may have a large number of additional internal states. If those internal states are not hidden using Promela's `atomic` construct, an explosion of the state space can ensue. Seasoned Promela users will not fall into this trap, but especially newcomers and students are likely to encounter this problem.

Another reason, which we have examined in Section 3, is that SPIN's fast but naive depth-first search fares very badly in the context of debugging systems with a very large (or infinite) state space. The mixed depth-first and breadth-first strategy of PROB can give much better results in such a setting.

Finally, in Section 4, we have shown that by exploiting symmetries in a high-level model, the model checking time can be dramatically reduced. For two examples, the PROB model checker performs verification in substantially less time than SPIN with partial order reduction and bitstate hashing.

Looking to the future, we believe there is a big potential for applying more intelligent model checking techniques to high-level formalisms. In particular, we believe that the potential for techniques such as heuristics-directed or parallel model checking is much more pronounced for a high-level formalism such as B than for a low-level formalism such as Promela.

In conclusion, due to the inherent exponential blow-up of the state space, it is often not that relevant whether a model checking tool can treat 100,000 or 10,000,000 states; it can be much more important how cleverly the tool treats those states and whether it can limit the exponential blow-up through techniques like symmetry reduction.

**Acknowledgements** We would like to thank Alastair Donaldson, Alice Miller, Daniel Plagge, Harald Wiegard, and Dennis Winter for insightful comments and contributions to this paper.

## References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. Arvind, N. Dave, and M. Katelman. Getting formal verification into design flow. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *Proceedings FM'08*, LNCS 5014, pages 12–32. Springer, 2008.
3. D. A. Basin, S. Friedrich, M. Gawkowski, and J. Posegga. Bytecode model checking: An experimental analysis. In Bosnacki and Leue [8], pages 42–59.
4. M. Ben-Ari. *Principles of Concurrent and Distributed Programming (Second edition)*. Addison-Wesley, 2006.
5. M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.
6. D. Bosnacki, D. Dams, and L. Holenderski. Symmetric Spin. *STTT*, 4(1):92–106, 2002.
7. D. Bosnacki, D. Dams, L. Holenderski, and N. Sidorova. Model checking SDL with Spin. In S. Graf and M. I. Schwartzbach, editors, *TACAS*, LNCS 1785, pages 363–377. Springer, 2000.
8. D. Bosnacki and S. Leue, editors. *Model Checking of Software, 9th International SPIN Workshop, Grenoble, France, April 11-13, 2002, Proceedings*, LNCS 2318. Springer, 2002.
9. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170, Jun 1992.
10. J. Chen and H. Cui. Translation from adapted uml to promela for corba-based applications. In S. Graf and L. Mounier, editors, *SPIN*, LNCS 2989, pages 234–251. Springer, 2004.
11. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
12. A. F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *Proceedings FM'05*, LNCS 3582, pages 481–496. Springer, 2005.
13. A. F. Donaldson and A. Miller. Exact and approximate strategies for symmetry reduction in model checking. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, LNCS 4085, pages 541–556. Springer, 2006.
14. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *STTT*, 6(4):277–301, 2004.
15. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with hsf-spin. In M. B. Dwyer, editor, *SPIN*, LNCS 2057, pages 57–79. Springer, 2001.
16. Formal Systems (Europe) Ltd. *Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)*.
17. P. Godefroid, editor. *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, LNCS 3639. Springer, 2005.
18. N. Guelfi and A. Mammar. A formal semantics of timed activity diagrams and its promela translation. In *APSEC*, pages 283–290. IEEE Computer Society, 2005.

19. G. J. Holzmann. The model checker Spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.
20. G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.
21. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
22. G. J. Holzmann and D. Peled. An improvement in formal verification. In D. Hogrefe and S. Leue, editors, *FORTE*, volume 6 of *IFIP Conference Proceedings*, pages 197–211. Chapman & Hall, 1994.
23. D. Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, 2002.
24. M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.
25. M. Leuschel and M. Butler. Automatic refinement checking for B. In K.-K. Lau and R. Banach, editors, *Proceedings ICFEM'05*, LNCS 3785, pages 345–359. Springer-Verlag, 2005.
26. M. Leuschel, M. Butler, C. Spermann, and E. Turner. Symmetry reduction for B by permutation flooding. In *Proceedings B2007*, LNCS 4355, pages 79–93, Besancon, France, 2007. Springer-Verlag.
27. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
28. M. Leuschel and T. Massart. Efficient approximate verification of B via symmetry markers. *Proceedings International Symmetry Conference*, pages 71–85, Januar 2007.
29. K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Boston, 1993.
30. D. Peled. Combining partial order reductions with on-the-fly model-checking. In D. L. Dill, editor, *CAV*, LNCS 818, pages 377–390. Springer, 1994.
31. A. Prigent, F. Cassez, P. Dhaussy, and O. Roux. Extending the translation from sdl to promela. In Bosnacki and Leue [8], pages 79–94.
32. G. Rothmaier, T. Kneiphoff, and H. Krumm. Using Spin and Eclipse for optimized high-level modeling and analysis of computer network attack models. In Godefroid [17], pages 236–250.
33. C. Spermann and M. Leuschel. ProB gets nauty: Effective symmetry reduction for B and Z models. In *Proceedings Symposium TASE 2008*, pages 15–22, Nanjing, China, June 2008. IEEE.
34. E. Turner, M. Leuschel, C. Spermann, and M. Butler. Symmetry reduced model checking for B. In *Proceedings Symposium TASE 2007*, pages 25–34, Shanghai, China, June 2007. IEEE.
35. B. D. Wachter, A. Genon, T. Massart, and C. Meuter. The formal design of distributed controllers with $_{\mathrm{d}}$sl and Spin. *Formal Asp. Comput.*, 17(2):177–200, 2005.
36. H. Wiegard. A comparison of the model checker ProB with Spin. Master's thesis, Institut für Informatik, Universität Düsseldorf, 2008. To appear.
37. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *DAC*, pages 599–604, 1998.

## A scheduler.prom

The following is a manual translation (and slight simplification) of the scheduler1.ref machine into Promela (with 2 processes).

```
chan readyq = [2] of { byte } ; bool activef=0;
proctype user () {
    bool created=0; bool idle=0; bool ready=0; bool act=0;
    label1:
    do
      :: atomic{(created==0) -> created = 1; idle = 1}
      :: atomic{(created==1 && idle==1) -> created = 0; idle=0}
      :: atomic{idle==1 ->  idle=0; ready=1; label2:readyq!_pid }
      :: atomic{(readyq?[eval(_pid)] &&  ready==1 && activef==0) ->
                    readyq?eval(_pid);ready = 0 ->
                    activef=1 ->  act = 1 }
      :: atomic{act==1 -> idle = 1; act = 0; activef = 0}
    od;
 }
 /* initialize flags and start the processes */
init {  atomic{  run user();  run user(); }; printf("init\n")}
```

## B Scheduler1.ref

```
REFINEMENT scheduler1_improved
REFINES scheduler0
VARIABLES proc, readyq, activep, activef, idleset
INVARIANT proc : POW(PROC) &  /* created */
          readyq : seq(PROC) & activep : POW(PROC) &
          activef : BOOL &  idleset : POW(PROC)
INITIALISATION
    proc:={}  ||  readyq:={} ||
    activep:={} || activef := FALSE || idleset := {}
OPERATIONS
 new(p) = PRE p : PROC - proc THEN
        idleset := idleset \/ {p} || proc := proc \/ {p} END;
 del(p) = PRE p : PROC & p : idleset THEN
        proc := proc-{p} || idleset := idleset - {p} END;
 ready(p) = PRE  p : idleset THEN
        readyq:=readyq<-p || idleset := idleset - {p} END;
 enter(p) =  PRE p : PROC & readyq/=<> &
                 p = first(readyq) & activef=FALSE THEN
        activep:={p} || readyq := tail(readyq) ||
        activef:=TRUE END;
 leave(p) = PRE p : PROC &  activef=TRUE & p : activep THEN
        idleset := idleset \/ {p} || activef := FALSE ||
        activep := {} END
END
```