p2b: A Translation Utility for Linking Promela and Symbolic Model Checking (Tool Paper)

Michael Baldamus[1] and Jochen Schröder–Babo

University of Kalsruhe, Institute for Computer Design and Fault Tolerance
P.O. Box 6980, 76128 Karlsruhe, Germany
{baldamus,schrbabo}@ira.uka.de

Abstract. p2b is a research tool that translates Promela programs to boolean representations of the automata associated with them. These representations conform to the input syntax of the widely–used symbolic model checker SMV; it is then possible to verify the automata with SMV, as opposed to enumerative model checking with SPIN, the classical Promela verifier. SMV and SPIN are focussed on verifying branching or linear time temporal properties, respectively, and often exhibit different performance on problems that are expressible within both frameworks. Hence we envisage that p2b will provide the missing link in establishing a verification scenario that is based on Promela as modeling language, and where one chooses different logics and verification methods as needed. The present paper provides an introduction to p2b, a description of how it works and two benchmark examples.

1 Introduction

An important ingredient of model checking is an expressive language that can be used for model description. Such a language must have a precise semantics, yet it must also be suitable for its application domain and easy to use. Promela [7], the input language of the SPIN model checker [8], is an asynchronous concurrent modeling language. It naturally does have a precise semantics and it arguably fulfills the other criteria too. SPIN then performs enumerative model checking of linear time temporal properties (LTL) over Promela programs. Two major optimizations realized by SPIN are on–thy–fly state space traversal and partial order reduction; they shorten runtime often dramatically. Murφ [5] is another well–known enumerative model checker that contains several optimizations of the basic procedure. Successful applications in various practical fields have shown how powerful enumerative model checking can be.

That success, however, does not come in all cases. A property may hold, for instance, meaning that it does *a priori* not help that on–the–fly traversals often find counterexamples without visiting every state that is relevant. Moreover, partial order reduction may greatly reduce the number of relevant states, but the method requires specific

[1] Michael Baldamus's work is supported by the Deutsche Forschungsgemeinschaft within the Project Design and Design Methodology of Embedded Systems.

preconditions with regard to the way processes communicate with each other: the efficiency gain suffers the more communication relationships violate those conditions.

Another method besides the enumerative one is symbolic model checking [4,10]. Its basic idea consists of working with reduced ordered binary decision diagrams (BDDs, [3]) to represent finite automata and sets of states. The "secret" is partly that many systems with large state spaces can be represented with comparatively small BDDs; besides that, most algorithms on BDDs have moderate complexity. Hence it has been possible to verify practical examples whose state sets are astronomically large. The main applications of symbolic model checking have to date been in verifying synchronous digital hardware. There are, however, encouraging results on verifying also asynchronous and interleaved processes [6,1,9,2], as they are typical for software–like systems. This situation was the reason for us to develop p2b. The objective was to perform symbolic model checking on such systems and, at the same time, to profit from Promela's versatility as a modeling language.

Efficient symbolic model checkers are readily available. The easiest way to achieve the objective of p2b is therefore to translate Promela programs to boolean representations of the automata associated with them, as symbolic model checkers usually understand this kind of input. More specifically, p2b generates code that adheres to the input syntax of the well–known and widely–used symbolic model checker SMV [10].

With SPIN, p2b and SMV, we have carried out various experiments. They indicate that enumerative and symbolic model checking may indeed exhibit rather different efficiency when applied to the automaton of one and the same Promela program. Sometimes SPIN is significantly faster, sometimes SMV. (cf. Section 3). Another possible benefit from p2b consists of the fact that symbolic model checking is first and foremost concerned with branching time temporal properties, as opposed to the LTL world to which SPIN belongs. Hence we envisage that p2b will provide the missing link in establishing a verification scenario that is based on Promela as modeling language, and where one chooses different logics and verification methods as needed.

We have to mention that SMV starts the actual model checking procedure strictly after it has built the BDD that represents the automaton of the model under consideration. For this reason, it may be somewhat difficult to represent a dynamically evolving system of concurrent processes. Such systems, on the other hand, can easily be modeled with Promela with the help of the keyword `run`, which spawns a new process instance. p2b does not support this feature at the moment. In consequence, p2b accepts Promela models whose process instances — *proctype* instances in Promela terminology — can easily be determined before verification or simulation takes place. To our experience, many practical systems fall into this category, notably within the realms of embedded systems and communication protocols.

The remainder of the present paper is structured as follows: Section 2 describes the basics of how p2b works; Section 3 presents two benchmark examples, the dining philosophers problem and a mutual exclusion protocol over asynchronous channels; Section 4 briefly concludes the paper.

The p2b homepage is located at

`http://goethe.ira.uka.de/~baldamus/p2b`.

It is possible to download the package from a subpage there.

2 How p2b Works

p2b is a command line utility. It works in batch mode in the sense that a run consists of parsing a Promela program and generating ASCII output that conforms to the input syntax of the SMV model checker. The basic idea is to identify every proctype instance of the program. The automaton of each individual instance is described in isolation; by putting together these descriptions, the automaton of the program as a whole is described.

2.1 SMV Code Generated by p2b

Then the raw structure of the output will in general be as follows:

```
MODULE main
VAR
  ≪ declarations of current state variables ≫
INIT
  ≪ initialization of current state variables ≫
DEFINE
  ≪ boolean equations ≫
TRANS
  ≪ top expression ≫
SPEC
  ≪ temporal formula ≫
```

Only the `SPEC` part may be missing (see Section 2.1.4). In the sequel of this subsection, we briefly discuss each individual part.

2.1.1 Variables and Variable Initialization If *P* is the program, then the output represents the automaton of *P* employing *current state variables* and *next state variables* to encode automaton states. First of all there are boolean variables that mostly correspond to control flow locations of the proctype instances of *P* but may also have auxiliary roles. Besides that, there may be *data variables*, which correspond to data variables and channel entries in *P*. The ordinary, explicitly declared SMV variables of both kinds are the current state variables. For every such variable, say *x*, there is a unique next state variable, which appears in the SMV code as `next(`*x*`)`. p2b does not have to allocate any next state variable since SMV does that automatically. The current state variables are declared in the `VAR` part; their initial values are assigned in the `INIT` part. This assignment encodes the initial state of the automaton of *P*.

2.1.2 Boolean Equations The variable declarations and initializations are followed by a `DEFINE` part. This part contains a collection of equations of the form *identifier* := *boolean expression*. Every right hand side may contain state variables or identifiers

defined by other expressions. The collection of equations is essentially a bottom-up description of the automata of the proctype instances of *P*. To give an impression of that, let A be an active proctype in *P* that has k instances, $k \geq 1$, and let l_1, ..., l_n be the control flow locations in A, $n \geq 1$. Then there are equations of the form $A\_i\texttt{-}l_j\texttt{\_enabled-} :=$ *boolean expression* for all $i \in \{1, \ldots, k\}$ and all $j \in \{1, \ldots, n\}$; they become true iff the corresponding control flow location is enabled. There are also equations of the form $A\_i\texttt{-at-}l_j :=$ *boolean expression* for all $i \in \{1, \ldots, k\}$ and all $j \in \{1, \ldots, n\}$; these ones describe the local and global effects of a transition of A that starts at the respective l_j, referring to $A\_i\texttt{-}l_j\texttt{\_enabled-}$ on the right hand side. Furthermore, there are equations of the form

$$A\_i\texttt{-} := \bigvee_{j=1}^{n} A\_i\texttt{-at-}l_j$$

for all $i \in \{1, \ldots, k\}$, describing the entire automaton of the respective instance of A. Besides that, there are equations of the form $A\_i\texttt{-idle-} :=$ *boolean expression* for all $i \in \{1, \ldots, k\}$; their role is to describe the idling of the corresponding instance of A if another proctype instance is active.

2.1.3 Top Expression Apart from the equation system, there is a top expression, which puts all proctype instance automata together. This expression makes up the content of the TRANS part. To see what it basically looks like, let *pre*$_1$, ..., *pre*$_m$ be the identifier prefixes that correspond to proctype instances of *P*, $m \geq 1$. — An example of such a prefix is A_i, $i \in \{1, \ldots, k\}$. — Then the top expression has the form

$$\left(\bigvee_{i=1}^{m} \left(\textit{pre}_i\texttt{-} \bigwedge_{j \in \{1,\ldots,m\} \setminus \{i\}} \textit{pre}_j\texttt{-idle-} \right) \right) \vee \mathrm{Term},$$

where Term is an expression that describes the idling of the entire system once every proctype instance has terminated. The idea is that a transition of the automaton of *P*, as long as it has not terminated, involves a transition of the automaton of some proctype instance of *P* and, at the same time, leaves all other instances idle. This scheme works under the assumption that all channels have a capacity of at least one, meaning that there is no synchronization via channels. At the moment, p2b does indeed not support channels of capacity zero. To our experience, this restriction is not too severe in terms of what Promela models of practical systems can still be translated with p2b.

2.1.4 Optional Temporal Specification SMV can read temporal formulas that are to be verified, so p2b also allows the user to include such a temporal specification in the Promela code as a specific pragma ignored by SPIN. The pragma must appear at the end of the input file, and it must be of the form

`/*p2b: SPEC` $\ll$ temporal formula $\gg$ `*/`.

The formula contained in it will appear at the end of the output file behind the keyword SPEC. Non–trivial temporal properties will usually be formulated with the help of the variables from the VAR part.

2.1.5 Complexity of the Translation The complexity of the translation is quadratic in the number of proctype instances; the reason of that is the syntactic structure of the top expression in the `TRANS` part. The complexity of generating the output up to and including the `DEFINE` part is linear in the number of proctype instances.

2.2 Supported Constructs

p2b rejects every input rejected by SPIN. The current version of p2b *does not accept* every input accepted by SPIN either, since it does not support all Promela constructs. This situation is mostly due to the limited manpower that could be allocated to the p2b project; it is only to a small extent due to any principal difficulty in translating Promela in the way adopted for p2b, that is, by generating a boolean representation of the program automaton over current and next state variables. The constructs supported by the current version are as follows:

- **Data Types** The supported data types are `bit`, `bool`, `byte`, `short` and `int`.
- **Channels and Variables** Channels must have a capacity of at least 1 and must be global. Variables may be either global or local. The _–variable is supported.
- **Expressions** p2b supports numeric constants, the usual boolean and arithmetic operators, bracketing, variable access and the `empty`, `nempty`, `full`, `nfull` and `?[..]`–operators for channel polling including the `eval` constraint in the case of `?[..]`.
- **Elementary Statements** p2b supports `skip`, assignments, expressions that appear as statements, standard send and receive operations on channels including `eval`, `goto`, the `xs`, `xw` and `xu` declarations, `printf` and `assert`. Among these statements, `xs`, `xw` and `xu` declarations and `printf` do not affect symbolic model checking, so they are treated like `skip`. `assert` is also treated like `skip`, as this statement runs somewhat contrary to the paradigm of breadth–first search used in symbolic model checking. If the functionality of `assert` is desired, then it should mostly be possible to use a temporal specification instead (see Section 2.1.4).
- **Statement Constructors** p2b supports sequential composition, `if`..`fi`, `if`..`::`..`else`..`fi`, `do`..`od`, `do`..`::`..`else`..`od`, `atomic`, `unless` and `{`..`}`.
- **Labeling** p2b supports labels wherever SPIN permits labels in Promela code.
- **Proctypes** p2b supports active proctypes with and without numeric instantiators. Non–active proctypes — and `init`–sections — will be supported once `run` is supported (cf. Section 1).
- **`never`–Claims, `trace` and `notrace`** p2b ignores any `never`–claim as well as `trace` and `notrace`. Temporal properties to be verified by SMV should be specified using the pragma described in Section 2.1.4.
- **`#define`** p2b supports C–style macros.

2.3 Specifying Variable Ranges

Symbolic model checking is sometimes affected by the fact that the representation of data operations may lead to large BDD sizes. A prime example is multiplication, since

in its case every BDD representation must be exponential in the width of the data path. For this reason, symbolic model checking may be greatly helped if it is known to what extent the data variables of the program are utilized. p2b allows the user to supply such information by means of pragmas that are ignored by SPIN. Such a construct has the form

`/*p2b:` ≪ smallest possible value ≫ `..` ≪ highest possible value ≫ `*/`

and may occur behind the types `byte`, `short` and `int`. Its effect is that p2b generates syntax that instructs SMV to allocate just enough BDD variables to accommodate the specified range.

3 Benchmark Examples

We have studied several examples with p2b. In each case we have verified a Promela program or a class of such programs with SPIN and — after translation — with SMV. This section reports on our results with regard to two scalable examples, the dining philosophers problem and a mutual exclusion protocol over asynchronous channels. We used SPIN V. 3.4.3 and Cadence Berkeley SMV V. 08-08-00p3 on an 800 MHz Pentium III processor under Linux with 700 MB of available RAM.

3.1 The Dining Philosophers Problem

The model of the dining philosophers problem represents philosophers as proctypes and chop sticks as channels of capacity one. As chop sticks can be considered passive items, we contend that this kind of model is natural. Due to the ring topology of the problem, every channel is then shared by two proctypes, which both read from and sent to the channel. Figure 1 shows measurements obtained from a solution without deadlock; the deadlock–preventing component is a dictionary, which is also represented as a channel of capacity one. All proctypes initially try to read from that channel or from a channel representing a chop stick. Only one proctype can succeed in reading from the channel representing the dictionary and it will write back to this channel before trying to read from any other one. Furthermore, the initial read from a channel representing a chop stick is guarded by the condition that the channel representing the dictionary be empty.

The result of this experiment was that SMV was significantly faster than SPIN from seven philosophers onwards. Moreover, SPIN ran out of memory from ten philosophers onwards even when compression was turned on. This situation was probably due to the combination of two facts: first, the on–the–fly strategy could not bear fruit since the property to be verified always held; second, the exclusive send and the exclusive read condition with regard to channels are violated in all cases, so partial order reduction was less effective than usual.

3.2 A Mutual Exclusion Protocol over Asynchronous Channels

We have observed converse tendencies in the case of a mutual exclusion protocol over asynchronous channels (Figure 2). That protocol consists of n processes that communicate with an arbiter via channels a_i, b_i and c, $1 \leq i \leq n$, where each channel is of

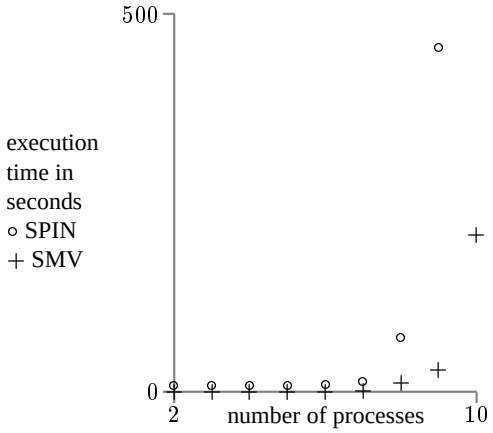

Fig. 1. Execution time measurements from verifying a deadlock–free solution to the dining philosophers problem. SMV was used via its graphical interface and the following options were set: Use heuristic variable ordering, Use modified search order, Restrict model checking to reachable states, Turn off transition relation clustering, Turn off conjunctively partitioned relations, Turn off partitioned relations. The other options of the graphical interface were not set.

capacity one. A process P_i sends its request for entering a critical section to the arbiter via a_i; the arbiters elects one such process, say P_j, and sends it a grant for entering the critical section via b_j. That process sends a notification via c to the arbiter once it has left the critical section.

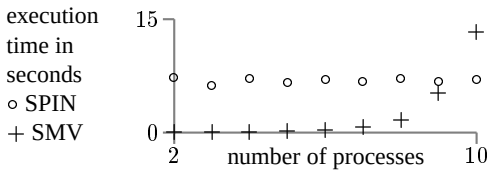

Fig. 2. Execution time measurements from verifying a mutual exclusion protocol over asynchronous channels.

As shown by the figure, SPIN's execution time remained virtually constant over the scaling range considered in the experiment. SMV's execution time, by contrast, was significantly higher for ten processes and showed a clear tendency to staying so for larger numbers of processes. This situation was probably due to the fact that nearly all communication relationships in the model satisfy the exclusive send or the exclusive read condition, meaning that partial order reduction could be effective.

4 Conclusion

The preceding sections have given an introduction to the objective of p2b, which consists of linking Promela and symbolic model checking. It was also described how p2b works and two scalable benchmark examples were presented. From the first example, we conclude that it indeed makes sense to supplement enumerative model checking of Promela programs with symbolic model checking; from the second example, however, we conclude that p2b will not entail that enumerative model checking is replaced by symbolic model checking.

As for future work, it would of course be desirable to extend the range of Promela constructs supported by p2b. Another topic might consist of incorporating results on the combination of partial order reduction and symbolic model checking [1,9].

References

1. R. Alur, R. Brayton, T. Henzinger, S. Qadeer, and S. Rajmani. Partial–Order Reduction in Symbolic State Space Exploration. In *Computer–Aided Verification*, pages 340–351. Springer–Verlag, 1997. Proceeding CAV '97.
2. M. Baldamus and K Schneider. The BDD Space Complexity of Different Forms of Concurrency, 2001. Accepted for ICACSD '01.
3. R. Bryant. Graph–Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
4. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990. Proceedings LICS '90 symposium.
5. D. Dill, A. Drexler, A. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992. IEEE Conference Proceedings.
6. R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for Symbolic Model Checking in CCS. In *Computer–Aided Verification*, LNCS 575, pages 203–213. Springer–Verlag, 1991. Proceedings CAV '91 conference.
7. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
8. G. Holzmann. The Model Checker SPIN. *IEEE Transactions on Computer Engineering*, 23:279–295, 1997.
9. Kurshan, R. and Levin, V. and Peled, D. and Yenigün, H. Static Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384, pages 345–357. Springer–Verlag, 1998. Proceedings TACAS '98 conference.
10. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.