The Model-Checking Kit[⋆]

Claus Schröter, Stefan Schwoon and Javier Esparza

Laboratory for Foundations of Computer Science,
University of Edinburgh,
email: {clau0603,schw1201,jav}@dcs.ed.ac.uk

Abstract. The Model-Checking Kit [8] is a collection of programs which allow to model finite state systems using a variety of modelling languages, and verify them using a variety of checkers, including deadlock-checkers, reachability-checkers, and model-checkers for the temporal logics CTL and LTL [7].

1 Introduction

Research on automatic verification has shown that no single model-checking technique has the edge over all others in any application area. Moreover, it is very difficult to determine a priori which technique is the most suitable for a given model. It is thus sensible to apply different techniques to the same model. However, this is a very tedious and time-consuming task, since each algorithm uses its own description language. The Model-Checking Kit [8] has been designed to provide a solution to this problem in an academic setting, with potential applications to industrial settings.

There exist many different models for concurrent systems. Within the Kit we chose 1-safe Place/Transition nets as the basic model for the following two reasons: (i) They are a very simple model with nearly no variants. In contrast most other models have many different variants. For instance, communicating automata can be synchronous or asynchronous, and communication can be formalised in different ways. Process algebras have a wealth of different operators and semantics, and there also exist many different high-level net models. (ii) Many different verification techniques which can deal with 1-safe P/T nets are available. Since 1-safe P/T nets have a well-defined partial order semantics, partial order techniques like stubborn sets [21] and net unfoldings [18] can be applied (as a matter of fact, these techniques were originally introduced for Petri nets). Since a marking of a 1-safe P/T net is just a vector of booleans, symbolic techniques based on BDDs [5], like those implemented in SMV [17], can also be used. And, of course, the standard interleaving semantics of Petri nets allows to apply explicit state exploration algorithms, like those of SPIN [13].

For systems modelled in a language with a 1-safe net semantics, all the techniques listed above are in principle applicable. Since each of these techniques has both strengths and weaknesses, it would be highly desirable to apply them

[⋆] http://www7.in.tum.de/gruppen/theorie/KIT/

all and to compare the results. However, a user who wishes to employ two or more verification packages does not have an easy task. In particular, the packages have different input formats, and so the user is forced to enter input data multiple times, a rather tedious task and one prone to introduce errors and inconsistencies.

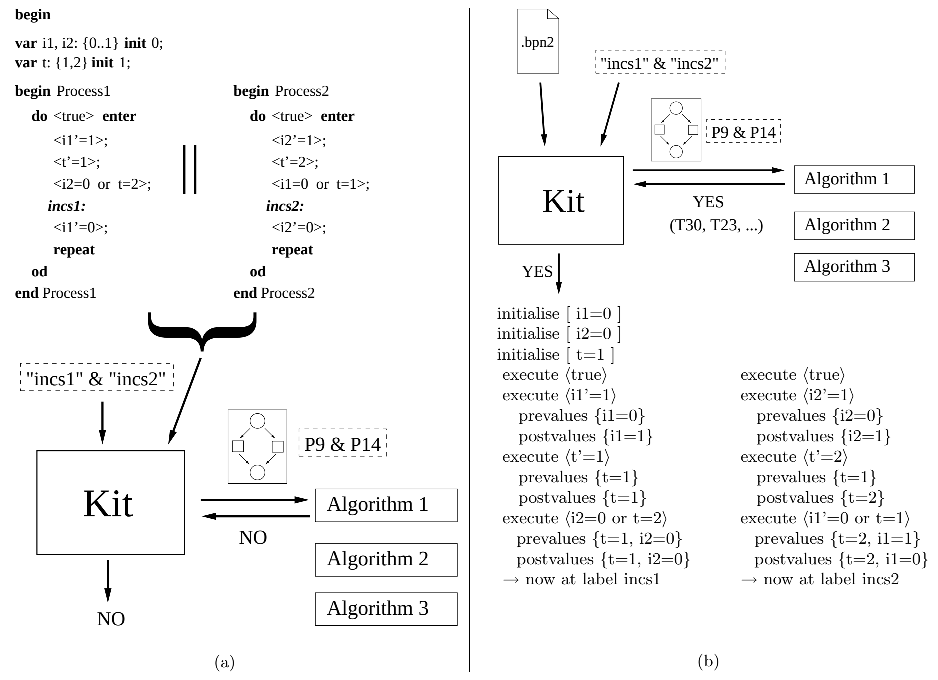
To amend this situation the Kit provides a shell which allows the user to specify input data (i.e. systems and properties) in a variety of input languages. Once a system and a property have been specified, the user can choose any of the model-checkers available in the shell to verify the property. The user is not required to be familiar with the different ways in which 1-safe P/T nets are represented to the different checkers.

The paper is structured as follows. In Section 2 we introduce a small example and use it to show how the Kit works. In Section 3 we present the modelling languages and verification techniques which are supported by the Kit. Section 4 gives a brief overview of the Kit's available options and their use. In Section 5 we present some experimental results which show performance differences of the individual verification techniques. Finally, we close with some conclusions in Section 6.

2 An Example

We show how the Kit works by means of a small example. We modelled Peterson's mutual exclusion algorithm [19] in B(PN)^2 as depicted in Figure 1 (a). B(PN)^2 [3], originally well-known from the PEP-tool [10], is a parallel programming language and one of the Kit's input languages. The notation is mostly self-explanatory, but for the sake of clarity we would like to point out two things: (i) ⟨`t'=1`⟩ means value assignment, whereas ⟨`t=1`⟩ means test of equality; (ii) `incs1:` denotes a label which can be used in formulae to mark a program point between two actions.

Suppose we want to check a mutual exclusion property, i.e. whether there exists a global system state in which both processes enter their critical sections simultaneously. The Kit allows this property to be expressed as `"incs1"` & `"incs2"`. The Kit takes the formula and the B(PN)^2 description and translates them into a 1-safe P/T net and a corresponding formula (e.g. `P9` & `P14`, where P9, P14 are place names of the net). Then the Kit invokes the model-checker chosen by the user (which would usually be one of the available reachability-checkers in this case). It checks whether there exists a reachable marking with tokens on both P9 and P14 simultaneously, and returns the result to the Kit. In case (a) the answer is 'no' which means that the mutex property holds. But what happens in case of an error? For that let us suppose that the user made a typo within the specification and wrote ⟨`i1'=0 or t=1`⟩ in Process 2 instead of ⟨`i1=0 or t=1`⟩. This causes a violation of the mutex property and the model-checker returns a transition sequence leading to the state which puts tokens simultaneously onto the places which represent the critical regions of the processes. As shown in Figure 1 (b) the Kit interprets this transition sequence at the



```
begin
var i1, i2: {0..1} init 0;
var t: {1,2} init 1;
begin Process1                    begin Process2
   do <true> enter                   do <true> enter
      <i1'=1>;                          <i2'=1>;
      <t'=1>;              ||           <t'=2>;
      <i2=0 or t=2>;                    <i1=0 or t=1>;
   incs1:                            incs2:
      <i1'=0>;                          <i2'=0>;
      repeat                            repeat
   od                                od
end Process1                      end Process2
```

```
initialise [ i1=0 ]
initialise [ i2=0 ]
initialise [ t=1 ]
 execute ⟨true⟩                           execute ⟨true⟩
 execute ⟨i1'=1⟩                          execute ⟨i2'=1⟩
   prevalues {i1=0}                         prevalues {i2=0}
   postvalues {i1=1}                        postvalues {i2=1}
 execute ⟨t'=1⟩                           execute ⟨t'=2⟩
   prevalues {t=1}                          prevalues {t=1}
   postvalues {t=1}                         postvalues {t=2}
 execute ⟨i2=0 or t=2⟩                    execute ⟨i1'=0 or t=1⟩
   prevalues {t=1, i2=0}                    prevalues {t=2, i1=1}
   postvalues {t=1, i2=0}                   postvalues {t=2, i1=0}
 → now at label incs1                     → now at label incs2
```

Fig. 1. Peterson's mutex algorithm in B(PN)^2

level of the chosen input language (i.e. B(PN)^2) and outputs the result suitably formatted to the user.

3 Modelling languages and verification techniques

In this section we briefly introduce the different modelling languages and verification techniques supported by the Kit. Furthermore, we give a quick overview of how to describe properties.

3.1 Modelling a system

The Kit offers several languages for modelling a system. These languages can be divided into so-called net languages and high-level languages which abstract from net details.

- **High-Level Languages**
  Loosely speaking, these description languages abstract from structural net concepts like places, transitions, and arcs. The Kit currently offers three such languages called **B(PN)^2**, **CFA**, and **IF**. B(PN)^2 [3] (Basic Petri Net Programming Notation) is a structured parallel programming language offering features such as loops, blocks, and procedures. It is well-known from

the PEP-tool [10]. CFA [9, 8] (Communicating Finite Automata) is a language for the description of finite automata which communicate via shared variables or channels of finite length. It offers very flexible communication mechanisms and is also one of the modelling languages of the PEP-tool [10]. Finally, IF [4] (Interchange Format) is a language proposed in order to model asynchronous communicating real-time systems. It is the common model description language of the European ADVANCE [1] (Advanced Validation Techniques for Telecommunication Protocols) project.

- **Net languages**
  The Kit supports two net languages, **PEP** and **SENIL**. In these languages one has to define places, transitions, and arcs explicitly. PEP [2] is the low level net language of the PEP-tool [10]. It is supported by the Kit mostly because some tools can automatically export models into this format. SENIL [8] (Simple Extensible Net Input Language) is designed to make it easy to specify small P/T nets by hand; it is suitable for small nets with at most a few dozens of nodes, but not for larger projects.

3.2 Describing properties

The Kit can be used to check several types of properties, e.g. deadlock-freeness, reachability, safety and liveness properties. Except for deadlock-freeness these properties will be expressed as formulae.

Reachability properties In our framework a reachability property is a statement about states of the system. For example, the mutual exclusion property of critical regions can be understood as a reachability property. It amounts to the question whether there exists a reachable global state of the system in which two processes enter their critical regions simultaneously. These properties can be expressed with so-called state formulae. A state formula is a propositional logic formula consisting of atomic propositions and logical operators.

Safety and liveness properties Safety and liveness properties are expressed as formulae of temporal logics like CTL and LTL. They are the most popular temporal logics, and together they can express all common safety and liveness properties. Here we give just a brief introduction to LTL and CTL according to [7].

- **LTL** means Linear-Time Temporal Logic. The underlying structure of time is a totally ordered set. Under the assumption that the time corresponds to $(\mathbf{N}, <)$, the time is discrete, has an initial moment with no predecessors and is infinite into the future. LTL formulae consist of atomic propositions, boolean connectives and temporal operators. Temporal operators are $\mathbf{G}p$ ("always p", "henceforth p"), $\mathbf{F}p$ ("sometime p", "eventually p"), $\mathbf{X}p$ ("nexttime p") and $p\,\mathbf{U}\,q$ ("p until q"). The Kit supports only the *next-free* fragment of LTL.

- **CTL**, meaning Computation Tree Logic, is a branching time logic. The underlying structure of time is assumed to have a branching tree-like nature. It corresponds to an infinite tree where each node may have finitely many successors and must have at least one successor. These trees have a natural correspondence with the computations of concurrent systems or nondeterministic programs. A CTL formula consists of a path quantifier [**A** (all paths), **E** (there exists a path)] followed by an arbitrary linear-time formula, allowing boolean combinations and nestings of linear-time operators (**G**, **F**, **X**, **U**).

3.3 Verification techniques

As mentioned in the introduction many different verification techniques for 1-safe Petri nets are available. These techniques include among others the explicit construction of the state space, stubborn sets [21], BDDs [5], and net unfoldings [18]. The explicit construction of the state space is the classical approach, and still adequate in cases where the state space explosion is not very acute. Stubborn sets are used to avoid constructing part of the state space. They exploit information about the concurrency of actions. Using symbolic techniques (e.g. BDDs) one can succinctly represent large sets of states. They can reach spectacular compactification ratios for regularly structured state spaces. Approaches which are based on unfolding techniques make use of an explicitly constructed partial-order semantics of the system. It contains information not only on the reachability relation, but also on causality and concurrency. This technique is adequate for systems exhibiting a high degree of concurrency.

When planning the Kit we intended to integrate various checkers such that each of the verification approaches mentioned above is represented by at least one checker. This has led to the following selection:

- The **PEP**-tool [10] (Programming Environment based on Petri nets) is a programming and verification environment for parallel programs written in B(PN)^2 or CFA. Programs can be formally analysed using methods which are based on the unfolding technique [18]. The PEP-tool is distributed by the Theory group (subgroup Parallel Systems) of the University of Oldenburg. PEP contributes to the Kit a deadlock-checker, a reachability-checker, and a model-checker for LTL.
- **PROD** [22] is an advanced tool for efficient reachability analysis. It implements different advanced reachability techniques for palliating the state explosion problem, including partial-order techniques like stubborn sets [21], and techniques which exploit symmetries. PROD is distributed by the Formal Methods Group of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology. PROD contributes to the Kit a deadlock-checker, a reachability-checker, a CTL- and LTL-checker.
- The **SMV** system [17] is a tool for checking finite state systems against specifications in the temporal logic CTL. The input language of SMV is designed to allow the description of finite state systems that range from

completely synchronous to completely asynchronous, and from the detailed to the abstract. SMV is distributed by the Carnegie Mellon University. Its verification algorithms are based on BDDs [5] and it contributes to the Kit a deadlock-checker and a CTL-checker.

- **SPIN** [13] is a widely distributed software package that supports the formal verification of distributed systems. It can be used as a full LTL model-checking system, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed and verified without the use of LTL. SPIN uses explicit construction of the state space. It is distributed by the Formal Methods and Verification Group of Bell Labs. SPIN contributes to the Kit an LTL-checker.
- The tool **MCSMODELS** [12] is a model-checker for finite complete prefixes (i.e. net unfoldings [18]). It currently uses the PEP-tool [10] to generate the prefixes. These prefixes are then translated into logic programs with stable model semantics, and the integrated Smodels solver is used to solve the generated problems. MCSMODELS is distributed by the Formal Methods and Logic Groups of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology. MCSMODELS contributes to the Kit a deadlock-checker and a reachability-checker.
- **CLP** [14] is a linear-programming model-checker. It uses net unfoldings [18] and can check among others deadlock-freeness, reachability, and coverability of a marking. CLP is distributed by the Parallelism Research Group of the University of Newcastle upon Tyne and contributes to the Kit a deadlock-checker.

4 How to use the Kit

In this section we show how to use the Kit for verification tasks. The Kit is a command-line oriented tool (called `check`) without any graphical user interface. The available options are listed by calling `check` without any arguments. Figure 2 shows an overview of its current version.

For a correct program call one has to type

```
check [options] <input>:<checker> <modelfile> <formulafile>
```

where

- `<input>` is a place holder for one of the available modelling languages, i.e. `cfa`, `bpn2`, `if`, `pep`, `senil`.
  Note: `<input>` may be omitted; in this case `check` guesses the input language by looking at the extension of `<modelfile>` (which should be `.cfa`, `.bpn2`, `.if`, `.ll_net`, or `.senil`, in the order of the languages mentioned above). The user is free to use arbitrary extensions, but then the language has to be specified explicitly.
- `<checker>` should be replaced by one of the available algorithms, see the list at the bottom of Figure 2.

```
Usage: check [options] <input>:<checker> <modelfile> <formulafile>
   or: check -r <name> [options] <checker> <formulafile>

     Options:
      -s <name>         save intermediate results under <name>
      -r <name>         resume from intermediate results saved under <name>
      -t <dir>          place temporary files in <dir> (default is '.')
      -v                run in verbose mode

     Available input formats:
      cfa               concurrent finite automata
      bpn2              B(PN)^2 language
      if                IF language
      pep               PEP low level net format
      senil             SENIL net format

     Available algorithms:
      CTL         : prod-ctl, smv-ctl
      LTL         : prod-ltl, pep-ltl, spin-ltl
      Deadlock    : prod-dl, smv-dl, pep-dl, mcs-dl, clp-dl
      Reachability: prod-reach, pep-reach, mcs-reach
```

Fig. 2. The Kit's available options

- `<modelfile>` is the name of the file containing the system specification.
- `<formulafile>` is the name of the file containing the formula to be checked. Note: For deadlock-checking no formula file is needed.
- `[options]` are as follows:
  - `-s <name>`
    Temporary files representing intermediate results will be saved in a tar-archive `mckit_save_<name>.tar`. Some algorithms profit from the reuse of intermediate results. For example, if one uses a method based on the unfolding technique for verifying many properties on the same system, it is sensible to calculate the unfolding only once and not for every property over and over again. So the unfolding can be saved with this option for reuse (see option `-r`).
  - `-r <name>`
    With this option one can reuse intermediate results saved before with option `-s <name>`. This is sensible if one wants to check many properties on the same system. Then the translation from the modelling language into the correct input format for the checker should be done only once and not for every single property. When using this option one should omit the modelling language and the modelfile. Then the correct program call is:
    ```
    check -r <name> [options] <checker> <formulafile>
    ```
    The selected checker can then take advantage of the files saved in the file `mckit_save_<name>.tar`.

	prod-dl	smv-dl	pep-dl	mcs-dl	clp-dl
peterson	7.04 (0.09)	0.24	0.04	0.05	0.03
plate(5)	46.68 (1.38)	*mem*	4.80	0.53	0.54
client/server	61.06 (0.79)	111.80	0.76	0.54	0.55
key(4)	37.63 (0.20)	*mem*	*mem*	*mem*	*mem*
fifo(30)	36.74 (0.72)	*mem*	*mem*	*mem*	*mem*

Fig. 3. Results for Deadlock-Checking

5 Experimental results

In this section we compare the performances of the algorithms by means of experimental results on several systems. The results demonstrate the point we made in the beginning, namely that no single method has the edge over all others. We present results for checking deadlock-freeness and some safety properties.

All experiments were performed on a Linux PC with 64 MByte of RAM and a 230 MHz Intel Pentium II CPU. The times are measured in seconds. The systems we used are as follows:

– peterson: Mutual exclusion algorithm [19].
– plate(5): Production cell which handles 5 plates [11, 15].
– client/server: Client/Server system with 2 clients and 1 server [1].
– key(4): Manages keyboard/screen interaction in a window manager for 4 customer tasks [6].
– fifo(30): 1-bit-FIFO with depth 30 [16, 20].

The systems are modelled in different languages. Peterson's mutual exclusion algorithm is modelled in B(PN)^2, and the client/server system in IF. All other examples are modelled in PEP's low-level net format.

Figure 3 shows the results for deadlock-checking. We split PROD's verification times for the following reason: At first, PROD reads the net description file and produces a corresponding C file. Then this C file is compiled and linked to an executable reachability graph generator program. Finally, the actual verification task is done by performing this executable program. Since PROD spends most of the time for the generation of the executable file, the pure verification times are quoted in parentheses.

Peterson's mutual exclusion algorithm is a small example, and all verification techniques behave well. But a look at the systems plate(5) and client/server already shows a big difference in the performances. The unfolding based techniques outperform PROD and SMV here. Actually, SMV runs out of memory during the verification of the production cell (signified by '*mem*'). On the contrary, the systems key(4) and fifo(30) are examples in which PROD beats the other tools.

The results for safety properties are depicted in Figure 4. The properties were expressed as LTL-formulas, and they were checked using LTL checkers. For the production cell we checked a mutual exclusion property: exactly one of

	prod-ltl	pep-ltl	spin-ltl
plate(5)	67.22 (2.52)	1.20	*mem*
client/server	257.57 (30.83)	2.51	20.74
key(4)	4206.00 (4152.66)	*mem*	11.53

Fig. 4. Results for LTL-Checking

three places of the net carries a token. For the client/server system we checked that a buffer overflow can not occur. For the key(4) system we checked a mutual exclusion property for two places. A look at the results confirms that the checkers behave quite differently here as well. We are able to verify the property for the production cell with PROD and PEP, but not with SPIN. On the contrary, for the client/server system SPIN checks the formula much faster than PROD. The key(4) system is an example which can be verified quickly with SPIN, but not with PEP.

It is important to notice that the performance of each of the tools may degrade a lot when used as part of the Kit. For instance, it is easy to model a system in PROMELA such that checking some easy reachability property with SPIN takes virtually no time; if the same system is modelled in, say **CFA**, and then checked again with SPIN, the verification can run out of memory. The reason is that the Kit transforms the initial CFA model into a 1-safe Petri net, and this net into PROMELA. If the model is data intensive, the net (and with it the PROMELA model) can easily blow-up. Another reason for a degraded performance is that the user of the Kit does not have access to all the flags of each of the checkers (future versions of the Kit should include this feature).

6 Conclusions

The Model-Checking Kit is a collection of programs which allow to model a finite-state system using a variety of modelling languages, and (attempt to) verify it using a variety of checkers. It has been successfully applied in a lab course on automatic verification. Special care has been taken to design it in a modular way and to make it easy to use and easy to install. Experiments with beta-testers have shown that a moderately skilled user can install the tool and verify the first property of a small system within half an hour. Furthermore, the Kit has a high degree of portability since all programs are written in plain C and we do not offer any graphical user interface. The Kit can be used for comparing the performances of different verification methods. However, it must be emphasised that, since each of the Kit's checkers has been optimised for its own modelling language, the Kit's internal language conversions can lead to important losses in performance. Finally, the Kit is an open library. Due to its modular design it is easy to add new description languages or checkers, and to replace old versions of checkers by new ones. Anyone who is interested in adding new languages and/or tools is cordially invited to contact the authors.

References

1. ADVANCE - Advanced Validation Techniques for Telecommunication Protocols. http://verif.liafa.jussieu.fr/∼haberm/ADVANCE/main.html.
2. E. Best and B. Grahlmann. PEP Documentation and User Guide 1.8. Universität Oldenburg, 1998.
3. E. Best and R. P. Hopkins. $B(PN)^2$ - a Basic Petri Net Programming Notation. In *PARLE'93*, LNCS 694, pages 379 – 390. Springer-Verlag, 1993.
4. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, L. Mounier, J. P. Krimm, and J. Sifakis. The Intermediate Representation IF. Technical Report. Vérimag, 1998.
5. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677 – 691, Aug. 1986.
6. J. C. Corbett. Evaluating Deadlock Detection Methods, 1994.
7. E. A. Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science*, volume B, pages 997 – 1067. Elsevier Science Publishers B. V., 1990.
8. J. Esparza, C. Schröter, and S. Schwoon. The Model-Checking Kit. http://www7.in.tum.de/gruppen/theorie/KIT/.
9. B. Grahlmann, M. Möller, and U. Anhalt. A new Interface for the PEP-tool - Parallel Finite Automata. 2nd Workshop of Algorithms and Tools for Petri nets. Oldenburg, 1995.
10. B. Grahlmann, S. Römer, T. Thielke, B. Graves, M. Damm, R. Riemann, L. Jenner, S. Melzer, and A. Gronewold. PEP: Programming Environment Based on Petri Nets. *Hildesheimer Informatik Berichte*, (14), May 1995. Universität Hildesheim.
11. M. Heiner and P. Deusen. Petri net based qualitative analysis - A case study. Technical report I-08/1995. Brandenburg Technische Universität Cottbus, 1995.
12. K. Heljanko. *Combining Symbolic and Partial Order Methods for Model Checking 1-Safe Petri Nets*. PhD thesis, Helsinki University of Technology, 2002.
13. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
14. V. Khomenko. CLP. http://www.cs.ncl.ac.uk/people/victor.khomenko/home.formal/tools/tools.html.
15. C. Lewerentz and T. Lindner. Formal Development of Reactive Systems: Case Study Production Cell. LNCS 891. Springer-Verlag, 1995.
16. A. J. Martin. Self-timed FIFO: An exercise in compiling programs into VLSI circuits. In *From HDL Descriptions to Guruanteed Correct Circuit Designs*, pages 133 – 153. Elsevier Science Publishers, 1986.
17. K. L. McMillan. *Symbolic Model Checking - An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.
18. K. L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *CAV'92*, LNCS 663, pages 164 – 174. Springer-Verlag, 1992.
19. M. Raynal. Algorithms For Mutual Exclusion, 1986.
20. O. Roig, J. Cortadella, and E. Pastor. Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets. In *ATPN'95*, LNCS 935, pages 374 – 391. Springer-Verlag, 1995.
21. A. Valmari. On-the-Fly Verification with Stubborn Sets. In *CAV'93*, LNCS 697, pages 397 – 408. Springer-Verlag, 1993.
22. K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. *Technical Reports*, B(13):1 – 56, Aug. 1995.