The Model-Checking Kit*

Claus Schröter, Stefan Schwoon and Javier Esparza**

Institutsverbund Informatik, Universität Stuttgart
email: {schroecs,schwoosn,esparza}@informatik.uni-stuttgart.de

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.

Out of the many different models for concurrent systems, we chose 1-safe Place/Transition nets as the basic model for the Kit for the following reasons: (i) They are a very simple model with nearly no variants, in contrast to most other models. 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 exist many different high-level net models. (ii) Many different verification techniques which can deal with 1-safe P/T nets are available. Since 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 (finite-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

* http://www7.in.tum.de/gruppen/theorie/KIT/

** Supported in part by EPSRC grant R93346/01 (An Automata-Theoretic Approach to Software Model Checking).

both strengths and weaknesses, it would be highly desirable to apply them all and to compare the results. However, employing two or more verification packages is not easy. In particular, the packages have different input formats, forcing the user to enter input data multiple times, a tedious and error-prone task.

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 Kit has been implemented as a command-line oriented tool without any GUI. While such a design may lessen the appeal of the Kit (especially to novices), it also has advantages: The Kit is easy to set up, can be run on many different platforms, and the incremental effort for adding further input languages and model checkers is small.

The paper is structured as follows. In Section 2 we show how the Kit works with a small example. In Section 3 we present the modelling languages and verification techniques supported by the Kit. Section 4 gives a brief overview of the available options in the Kit. In Section 5 we present some experimental results showing how the verification techniques differ in performance, before concluding in Section 6.

2 An Example

Figure 1 (a) shows Peterson's mutual exclusion algorithm [19] modelled in the language $B(PN)^2$, which is a parallel programming language developed as part of the PEP tool [10] 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 the system can reach 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 translates the $B(PN)^2$ model and the formula into a 1-safe P/T net and a corresponding formula (e.g. `P9` & `P14`, where P9, P14 are places in the net). Then the Kit invokes the model checker chosen by the user; in this case, the user would typically choose one of the reachability checkers. The model checker examines whether there exists a reachable marking with tokens on both P9 and P14 simultaneously and returns the result to the Kit. In Figure 1 (a) the answer is 'no', which means that the mutex property holds. What, then, happens in case of an error? Let us suppose that the user made a mistake and wrote ⟨`i1'=0 or t=1`⟩ in Process 2 instead of ⟨`i1=0 or t=1`⟩. This causes the mutex property to become violated, and the model checker returns a transition sequence to a state with tokens in both of the places representing the critical

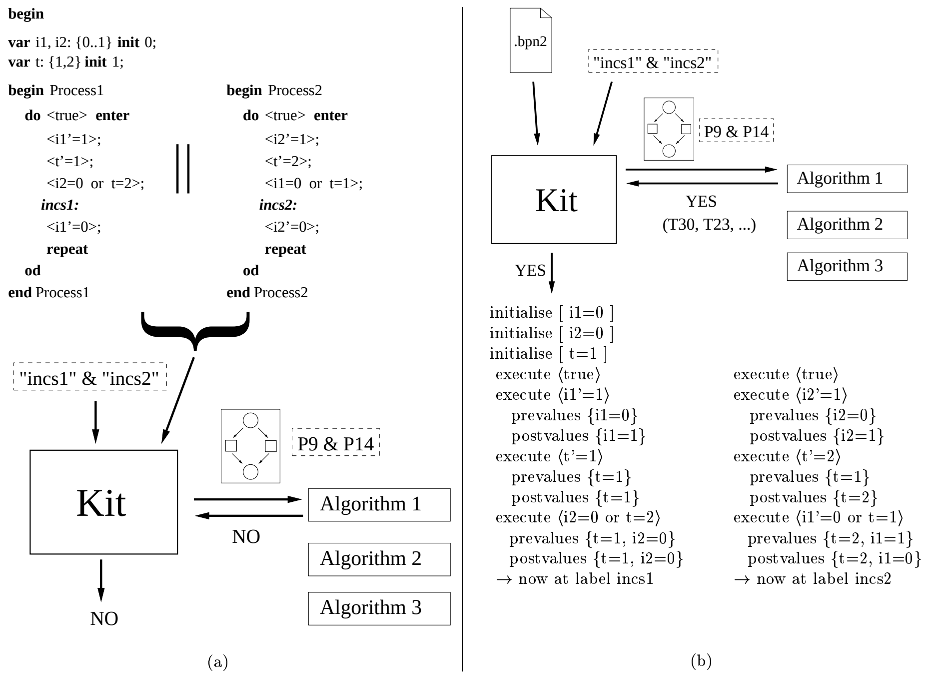

Fig. 1. Peterson's mutex algorithm in $B(PN)^2$

regions. As shown in Figure 1 (b), the Kit interprets this transition sequence at the level of $B(PN)^2$ and outputs the result 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, and how properties are described.

3.1 Modelling a system

The languages offered by the Kit for modelling a system can be divided into so-called net languages and high-level languages that 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: $\mathbf{B(PN)^2}$, **CFA**, and **IF**. $B(PN)^2$ [3] (Basic Petri Net Programming Notation) is a structured parallel programming language with features such as loops, blocks, and procedures used in the PEP tool [10]. CFA [9, 8] (Communicating Finite Automata) is a language for describing finite automata that 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] project. All three languages have a 1-safe Petri net semantics.

- **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.
  Users can use all available model checkers provided their nets are 1-safe. Whether a certain model checker can be used on non-1-safe nets depends on the tool in question (e.g., the translation into SMV fails in this case).

An interesting feature of the Kit is that, independently of the description language chosen by the user, all checkers can be applied to the same model. The counterexamples produced by the checker are presented to the user in terms of the description language used to model the system.

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 checkers (which always check the property of deadlock freeness) the model checkers expect a formula describing the property of interest. Formulae consist of atomic proposition and various logical or temporal connectives. The precise syntax and semantics of atomic propositions depends on the modelling language, e.g. net languages allow properties expressing that certain places are marked, whereas high-level languages allow to speak of properties of variables, channels and so forth. The syntax for logical connectives has been unified, so that, e.g., the same CTL formula can be used on all available CTL checkers.

Reachability properties In our framework a reachability property is the question whether the system can reach a certain set of global states (the mutual exclusion property is an instance of a reachability property). Such a property can be expressed with a so-called state formula, a propositional logic formula consisting of atomic propositions and logical operators.

Temporal logics The Kit supports the two temporal logics LTL and CTL. These two are arguably 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** (Linear-Time Temporal Logic) assumes that the underlying structure of time is a totally ordered set such that the time is discrete, has an initial moment with no predecessors and is infinite into the future. LTL formulae consist of atomic propositions, and boolean and temporal connectives. Temporal operators are $\mathbf{G}p$ ("always p"), $\mathbf{F}p$ ("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** (Computation Tree Logic) is a logic whose 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 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 $\mathrm{B(PN)}^2$ 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] implements different advanced techniques for reachability analysis to mitigate the state explosion problem, including partial-order techniques like stubborn sets [21], and techniques that exploit symmetries. PROD is distributed by the Formal Methods Group of the Laboratory for Theoretical Computer Science at the Helsinki University of Technology. PROD con-

tributes to the Kit a deadlock checker, a reachability checker, and model checkers for CTL and LTL.
- 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 PEP [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 absence of deadlocks, reachability, and coverability of a marking. CLP is distributed by the Parallelism Research Group of the University of Newcastle upon Tyne and contributes 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. The Kit is called as follows:

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

- `<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 the Kit 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.

```
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 '.')
      -o <opt>      specify options to pass to checker
      -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

- `<checker>` should be replaced by one of the available algorithms, see the list at the bottom of Figure 2.
- `<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 (`-t` and `-v` should be self-explanatory):
  - `-s <name>`
    Temporary files representing intermediate results will be saved in a tar file by the name `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 compute the unfolding only once per system and not once for every property. This option allows the the unfolding to be saved for reuse (see option `-r`).
  - `-r <name>`
    With this option one can reuse intermediate results saved earlier with `-s`. 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, i.e. the calling syntax becomes:

	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

`check -r <name> [options] <checker> <formulafile>`

The selected checker can then take advantage of the files saved in the file `mckit_save_<name>.tar`.

- `-o <opt>`

  Many model checkers allow to influence their behaviour with a variety of options. The Kit uses every tool with a general-purpose set of options recommended by the authors, which is usually fine for educational purposes. Expert users of the Kit, however, may want to use the full power of the tools. The Kit passes the value of `<opt>` on in the call to the chosen model-checker, and its meaning depends on that tool.

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 were as follows:

- peterson: Mutual exclusion algorithm [19].
- plate(5): Production cell handling 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 because PROD generates a C program from the net description. The actual verification task is carried out by executing the resulting program. Since PROD spends most of its time generating the executable file, the pure verification times are quoted in parentheses.

	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

Peterson's algorithm is a small example, and all techniques behave well. The systems plate(5) and client/server, however, exhibit big differences in the performances. The unfolding-based techniques outperform PROD and SMV here. Actually, SMV runs out of memory during the verification of plate(5) (shown as '*mem*'). In contrast, PROD beats the other tools on key(4) and fifo(30).

Figure 4 shows results for safety properties. These properties were expressed as LTL formulae and checked using LTL checkers. For the production cell we checked a mutual-exclusion property: exactly one of 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 were able to verify the property for the production cell with PROD and PEP, but not with SPIN. On the other hand, SPIN handles the client/server example much faster than PROD. The key(4) system is an example that can be verified quickly with SPIN, but not with PEP.

Notice that the tools, when used as part of the Kit, may not perform as well as when used alone. It is possible that using SPIN to model-check a PROMELA system takes virtually no time, whereas, if the same system is modelled as a 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 hence the PROMELA model) can easily blow up.

6 Conclusions

The Model-Checking Kit is a collection of programs that 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 install. In our experience, a moderately skilled user can install the tool and verify the first property of a small system within half an hour. Moreover, the Kit is highly portable since all programs are written in 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 performance losses. Finally, the Kit is an open library. Due to its modular design it is easy to add new input languages or checkers, and to replace old versions of checkers by new ones. Anyone interested in adding new languages and/or tools is very welcome 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, 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. Technical Report 14, Universität Hildesheim, May 1995.
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. Case Study Production Cell. In *Formal Development of Reactive Systems*, LNCS 891. Springer, 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, 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, 1995.

21. A. Valmari. On-the-Fly Verification with Stubborn Sets. In *CAV'93*, LNCS 697, pages 397–408. Springer, 1993.
22. K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD Reference Manual. *Technical Reports*, B(13):1–56, Aug. 1995.