An Efficient Specification for Model Checking Using Check-Points Extraction Method

CHIKATOSHI YAMADA
Okinawa National College of Tech.
Dept. of Inf. Comm. Syst. Eng.
905 Henoko, Nago, Okinawa 905-2192
JAPAN

YASUNORI NAGATA
Univ. of the Ryukyus
Dept. of Electrical & Electronics Eng.
1 Senbaru, Nishihara, Okinawa 903-0213
JAPAN

Abstract: In design of complex systems, embedded systems, and other critical systems, model checking, explores a finite state space to determine whether or not a given property holds, has played an important role. However, it is inefficiency to verify the entire systems. This article considers the case where designers of systems can extract check-points easily in model checking of formal verification. Moreover, we demonstrate some verification results by SPIN and NuSMV model checking tools.

Key–Words: Model checking, Linear temporal logic, Check-points extraction method

1 Introduction

Today, industrial designs are becoming more and more complex as technology advances and demand for higher performance increases. Especially, hardware and software systems are widely used in applied field where no failure is permitted: telephone switched network, electronic commerce, medical equipment, and other critical systems. The validity of a design accompanies checking whether the physical design satisfies its specification. In traditional design flow, validation is accomplished through simulation and testing. Some errors inside a design may exhibit nondeterministic behaviors, and therefore, will not be reliably repeatable. This makes testing and debugging using simulation difficult. Also, exhaustive testing for nontrivial designs is generally infeasible, therefore, testing provides at best only a probabilistic assurance[1].

In design of complex systems, embedded systems, and other critical systems, model checking, explores a finite state space to determine whether or not a given property holds, has played an important role. Model checking of formal verification ascertains whether designed systems can be executed or specified. Various formal methods for verification have been studied[1, 2, 3, 4]. However, formal verification has problems of its own class too. The major problem with automatic formal verification is that a large amount of memory and time is often required, because the underlying algorithm in these methods usually involves systematic examination of all reachable states of the system to be verified. As the number of reachable states increases rapidly with the size of the system, the basic algorithm by itself becomes impractical: the number of states for the system is often too large to check exhaustively within the limited time and memory that is available. This phenomenon is known as the state space explosion problem[1, 2].

In this research, we focus on specification process of model checking in formal verification shown in **Fig.1**, and to propose a new method which can extract verification check-points inductively from modeling systems. System designers can easily derive check-points of verified systems by using the method. The rest of this article is organized as follows: In section 2, Model Checking, Signal Transition Graph, Temporal Logic are briefly explained, and in section 3 our proposed Check-Points Extraction Method is described by means of procedure of specification. Moreover, some models are used for verification to compare by SPIN and NuSMV model checking tool in section 4. Finally, we summarize the discussion in section 5.

2 Preliminaries

2.1 Model Checking

The principal validation methods for complex systems are simulation, testing, deductive verification, and model checking. Simulation and testing both involve making experiments before deploying the system, testing is performed on the actual product. In the case of circuits, simulation is performed on the design of the circuit, whereas testing is performed on

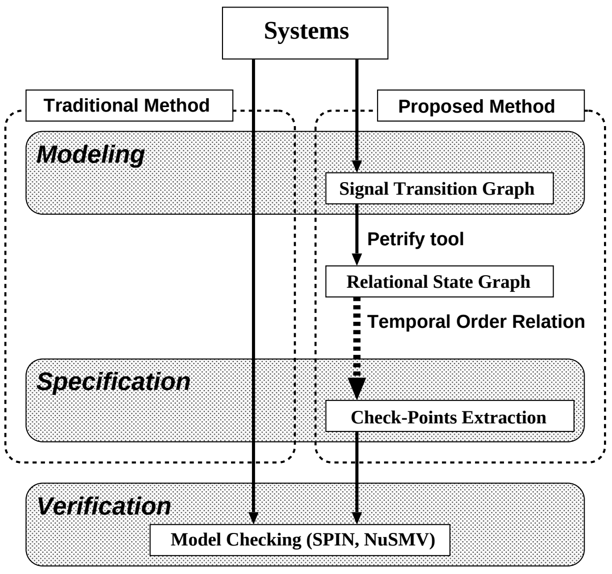


Figure 1: The framework of proposed method.

the circuit itself. In both cases, these methods typically inject signals at certain points in the system and observe the resulting signals at other points. These methods can be a cost-efficient way to find many errors. However, checking all of the possible interactions and potential pitfalls using simulation and testing techniques is rarely possible. Formal verification attempts to overcome the weakness of non-exhaustive simulation by proving the correspondence between some abstract specification and the design in hand.

An important issue in specifications completeness. Model checking provides means for checking that a model of the design satisfies a given specification, but it is impossible to determine whether the given specification covers all the properties that the system should satisfy. *Safety property* expresses that, under certain conditions, nothing bad *will happen*. *Liveness property* express that, under certain conditions, something good *will eventually happen*.

2.2 Signal Transition Graph

In order to describe highly concurrent systems, graph-based specification methods have been widely used. An Signal Transition Graph (STG)[6], a labeled interpreted Petri Net[7], has been considered as a well-suited specification method to describe asynchronous circuits.

Definition 1 (Petri Net (PN)). A *Petri Net* is a bipartite directed graph consisting of 4-tuple $\sum$ = (P, T, F, m_0), where

1. P is a finite set of places.
2. T is a finite set of transitions, satisfying $P \cap T = \phi$ and $P \cup T = \phi$.
3. F is a flow relation $F \subseteq (P \times T) \cup (T \times P)$, specifies binary relation between transitions and places.
4. m_0 is the initial marking of the PN.

When transitions are interpreted as rising and falling transitions of signals of a control circuit, an STG is one interpretation of a PN.

Definition 2 (Signal Transition Graph (STG)). Let J be a set of signals of a network, A *Signal Transition Graph* defined on J is a Petri Net $\sum_J$ = $\langle$ P, T, F, M_0 $\rangle$ with $T : J \rightarrow \{ + , - \}$.

Each transition of the STG is interpreted as a rising transition or a falling transition of a signal.

Consider an arbiter module shown in **Fig.2**. An STG for the arbiter module is shown in **Fig.3**, where '+' mean a rising edge and '-' means a falling edge of a certain signal, respectively. This example uses two signals **u0** and **u1**. Black circle on a transition edge indicates a token. A transition is enabled when all input places have at least one token. When an enabled transition fires, it removes one token from each input place and adds one token to each output place.

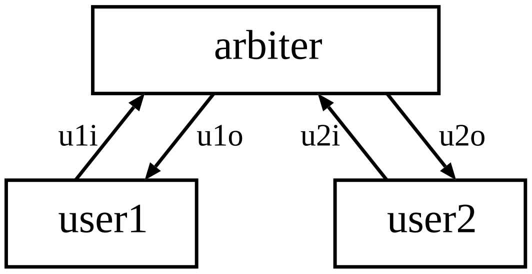


Figure 2: An arbiter module.

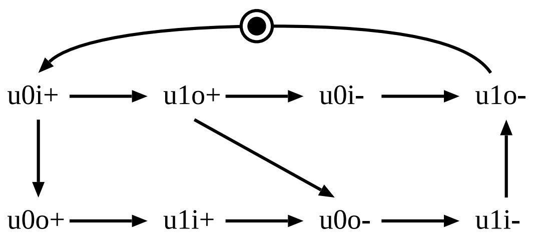


Figure 3: A signal transition graph for **Fig.2**

2.3 Temporal Logic

Temporal logic[1, 2, 4, 5] is a formalism for describing sequences of transitions between states in a reactive system. In the temporal logics that we will consider, time is not mentioned explicitly; instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special *temporal operators*. These operators can also be combined with boolean connectives or nested arbitrarily. Temporal logics differ in the operators that they provide and the semantics of those operators. Its operators mimic linguistic constructions (the adverbs ”always” , ”until” , the tenses of verbs, etc.) with the result that natural language statements and their temporal logic formalization are fairly close. Finally, temporal logic comes with a formal semantics, an indispensable specification language tool.

2.3.1 Linear Temporal Logic(LTL)

Temporal logic allows us to formalize the properties of a run unambiguously and concisely with the help of a small number of special temporal operators. Most relevant to the verification of asynchronous process systems is a specific branch of temporal logic that is known as linear temporal logic(LTL), commonly abbreviated as LTL. The semantics of LTL is defined over infinite runs. With help of the stutter extension rule, however, it applies equally to finite runs[1].

Here we give descriptions of LTL. LTL is a sort of temporal logic, which has the following formulas:

- $\Box\, q$: means that q always holds for all successor states on a certain path.
- $\Diamond\, q$: represents that q must be sometimes true for only one successor state of the path, and is similar to the formula which expresses future in linear temporal logic.
- $p\mathbf{U}q$: is that p must be true on the path states, beginning at the current state, until q becomes true.
- $\mathbf{X}p$: then simply states that p is true in the immediately following state of the run.

The correctness of properties to be verified is usually specified in LTL. The LTL is extending propositional logic with temporal operators that express how propositions change their truth values over time. Here we use temporal operators: Operators $\Box$, $\Diamond$, and $\mathbf{X}$ meaning *globally*, *sometime in the future*, and *next time*, respectively.

3 Check-Points Extraction Method

3.1 Strong/Weak Temporal Order Relation

In verifying behaviors of a system, checking all signal events is inefficient. Reducing signal events to be checked is necessary for specifying behaviors of the system. Here, We consider a system which has 3-inputs (a , b , c) and 2-outputs (x , y). Suppose that behaviors of the system occur as $a \rightarrow x \rightarrow b \rightarrow c \rightarrow y \rightarrow a$, repeatedly. All relations of the signal events can be indicated as follows:

$$\{(a\,,x)\,,(a\,,y)\,,(x\,,b)\,,(b\,,c)\,,(b\,,y)\,,(c\,,y)\},$$

where (a , x) indicates that output x occur after input a . Although output y is not an immediate successor of input a , (a, y) can be considered because output y must occur after input a in the future. Definitions of *strong/weak temporal order relations* are as follows:

Definition 3 (strong temporal order relation). *A strong temporal order relation is any inverse input-output relation of event sequences.*

Definition 4 (weak temporal order relation). *A weak temporal order relation is any relation of input signal events.*

Thus, behaviors of the system can be specified by introducing strong/weak temporal order relations as follows:

$$\{\ (a\,,x)\,,(a\,,y)\,,(b\,,y)\,,(c\,,y)\ \}$$

Its specification shows that output x can occur after input a and output y can occur by rendezvous inputs a, b, and c.

3.2 Converting STG to State Graph

To explain the procedure of the proposed method, we especially consider an arbiter module shown in **Fig.2**. Thus we describe specification of temporal formulas for the arbiter module. The STG of the arbiter module can be drawn in **Fig.3**. For the STG, states are connected with labeled edges as shown in **Fig.4** to represent order relations of events. Converting the STG to the state graph can be made by Petrify tool[8] automatically. A branch expression for **Fig.4** is shown in **Fig.5**. The procedure of the proposed specification method is described in the succeeding sections.

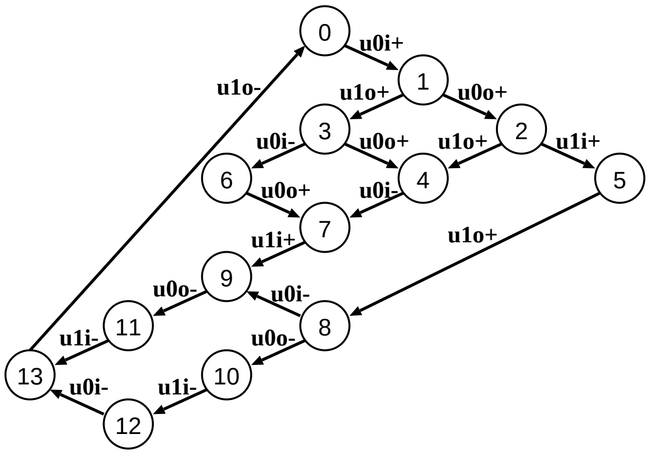


Figure 4: A state graph for **Fig.3**.

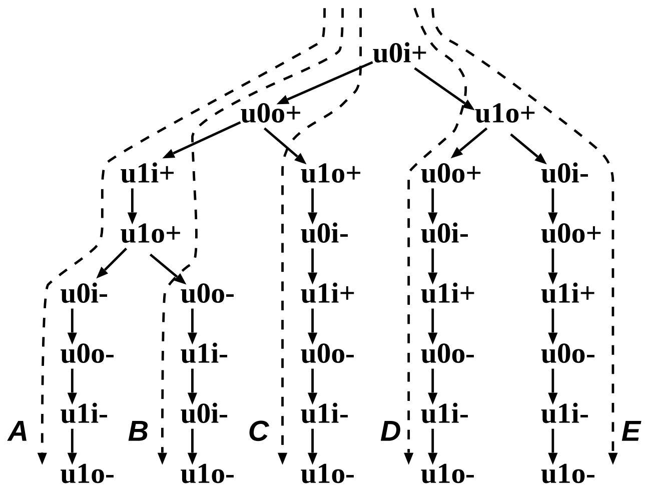


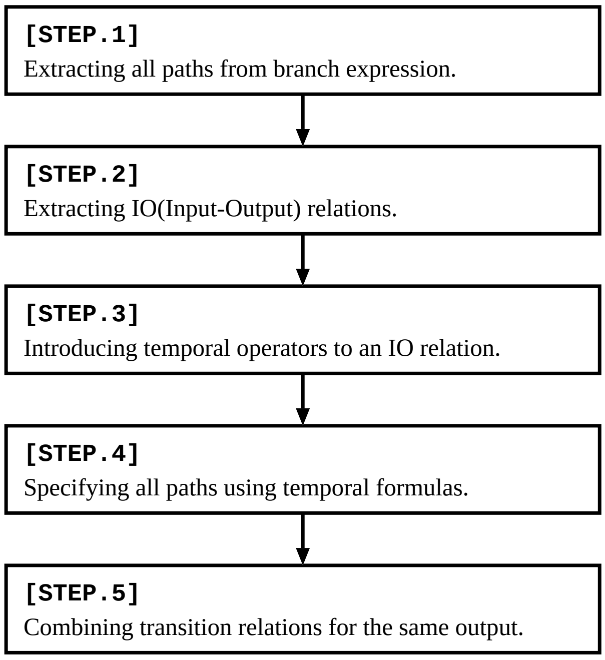
Figure 5: A branch expression for the state graph.

3.3 Procedure of Specification

In this section, we describe the procedure of the proposed specification method shown in **Fig.6**. This procedure corresponds to the part in the wavy arrow line in **Fig.1**. The procedure is composed of five steps shown in **Fig.6**. Here, we explain the procedure as follows:

[STEP.1]
In this step, event sequences are extracted from branch expression, for example, path (*A*), (*B*), (*C*), (*D*) and (*E*) are extracted from **Fig.5**.

(*A*) $u0_{i+}$ $u0_{o+}$ $u1_{i+}$ $u1_{o+}$ $u0_{i-}$ $u0_{o-}$ $u1_{i-}$ $u1_{o-}$
(*B*) $u0_{in+}u0_{o+}$ $u1_{i+}$ $u1_{o+}$ $u0_{o-}$ $u1_{i-}$ $u0_{i-}$ $u1_{o-}$
(*C*) $u0_{i+}$ $u0_{o+}$ $u1_{o+}$ $u0_{i-}$ $u1_{i+}$ $u0_{o-}$ $u1_{i-}$ $u1_{o-}$
(*D*) $u0_{i+}$ $u1_{o+}$ $u0_{o+}$ $u0_{i-}$ $u1_{i+}$ $u0_{o-}$ $u1_{i-}$ $u1_{o-}$
(*E*) $u0_{i+}$ $u1_{o+}$ $u0_{i-}$ $u0_{o+}$ $u1_{i+}$ $u0_{o-}$ $u1_{i-}$ $u1_{o-}$


[STEP.1]
Extracting all paths from branch expression.

[STEP.2]
Extracting IO(Input-Output) relations.

[STEP.3]
Introducing temporal operators to an IO relation.

[STEP.4]
Specifying all paths using temporal formulas.

[STEP.5]
Combining transition relations for the same output.

Figure 6: Procedure of Specification.

[STEP.2]
In this step, checked signal events can be reduced by introducing *strong/weak temporal order relations*.

[STEP.3]
In each path, if IO relation shows that there is immediate successor, specified as **X** operator, otherwise specified as $\Diamond$ operator.

[STEP.4]
In all paths, relations of the same temporal operator and the same IO can be extracted. Otherwise only the same IO relation can be extracted. Since $\Diamond$ expresses "*sometime in the future*," the *next* operator **X** can be covered as $\mathbf{X} \subseteq \Diamond$ in order to apply *Partial Order Reduction*. Thus, the extracted same IO relation can be gathered by $\Diamond$.

[STEP.5]
In all paths, relations of the same output can be combined.

$$\Box\ [\ \Diamond(u0_{i+}, u0_{o+}) \vee \Diamond(u0_{i+} \wedge u1_{i+}, u0_{o-}) \\ \vee \Diamond(u0_{i+} \wedge u1_{i+}, u1_{o+}) \vee \Diamond(u0_{i-} \wedge u1_{i+}, u1_{o-})]$$

Check-points can be extracted by repeating the above-mentioned steps.

4 Verification Results

In this section, we show verification results for a shared resources access structure shown in **Fig.7**. All these model verifications are performed on an 2.4GHz Core 2 Duo processor under Linux with 2GB of available RAM. In this article, all simulations are verified by SPIN version 4.2.9, XSPIN version 4.3.0[1, 3, 9, 10], and NuSMV version 2.4.3[11].

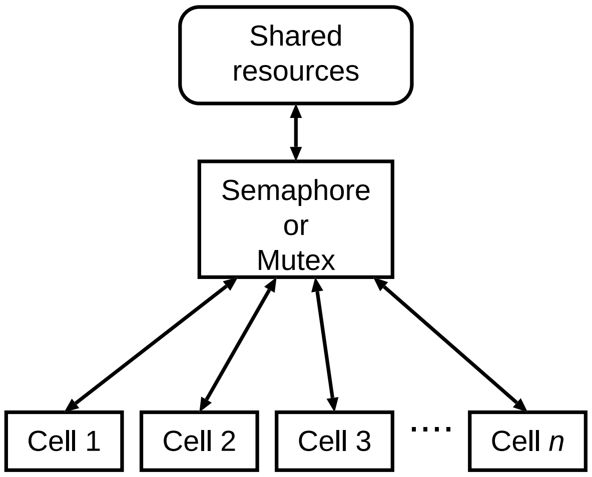


Figure 7: A shared resources access structure.

For the structure, we report the number of OBDD nodes necessary to represent the corresponding structure, transitions, and memory required by the systems to analyze the structure. Here, `CPE` indicates verification results with check-points extraction method, and `Normal` indicates verification results without the method, respectively. For verification of semaphore by SPIN shown in **Fig.8**, results are not much different between with and without the method. Similarly, verification results of mutex are also not much.

Moreover, performance results of semaphore and mutex by NuSMV are shown in **Fig.9** and **Fig.10**, where the number of cells refer to the number of cell modules shown in **Fig.7**. For the results of semaphore, verification cannot be improved very much. On the other hand, verification of mutex shows efficient results for more and more larger structures.

5 Conclusion

In design of complex systems, embedded systems, and other critical systems, model checking, explores a finite state space to determine whether or not a given property holds, has played an important role. However, it is inefficiency to verify the entire systems. This article considered the case where designers of systems can extract check-points easily in model checking of formal verification. Users must generally know well temporal specification because the specification might be complex. Our proposed method, check-points extraction method, can gain temporal

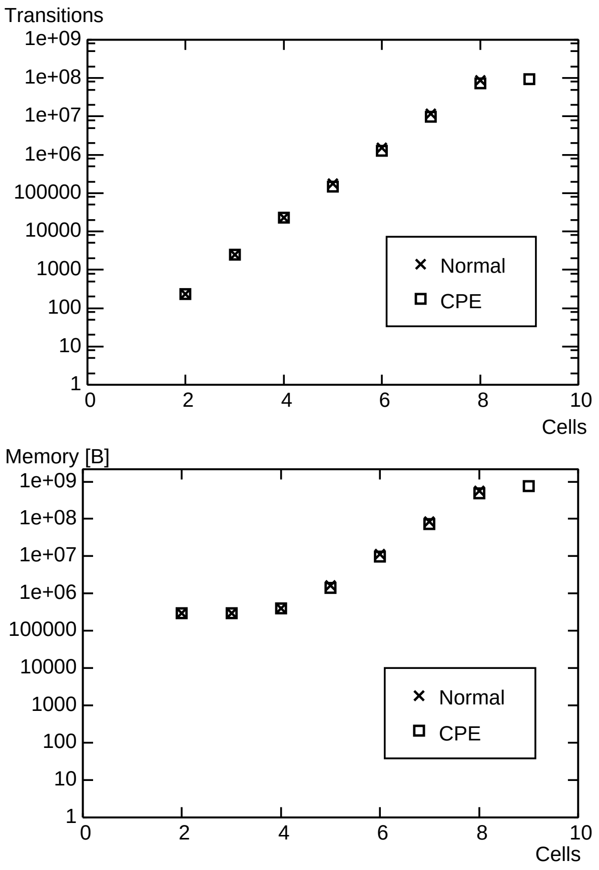


Figure 8: Verification performance of `semaphore` by SPIN: Transitions(upper), amount of Memories(lower).

formula specifications inductively. We aimed at input-output order relations for systems, not considering output-input order relations. Furthermore, we defined strong/weak temporal order relations in the procedure of specification. Weak temporal order relations include orders of inputs implicitly. Strong temporal order relations express inverse input-output order relations. We showed that the verification tasks are reduced for states, transitions, and memory with our proposed inductive specification method. System designers can easily lead complex temporal formulas by using the method. In verification results, especially, required memory was able to reduced for model checking. Then, it is assumed to be research work in the future to verify embedded systems design.

References:

[1] E.M. Clarke, O. Grumberg, and D. A. Peled: *Model Checking*, MIT Press, 2001.

[2] T. Kropf: *Introduction to Formal Hardware Verification*, Springer, 1999.

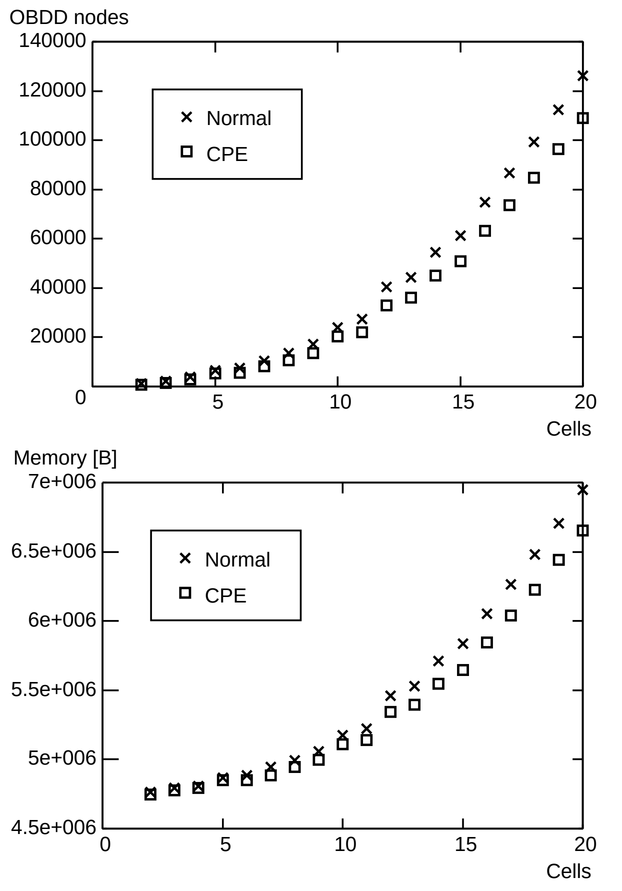


Figure 9: Verification performance of `semaphore` by NuSMV: OBDD nodes(upper), amount of Memories(lower).

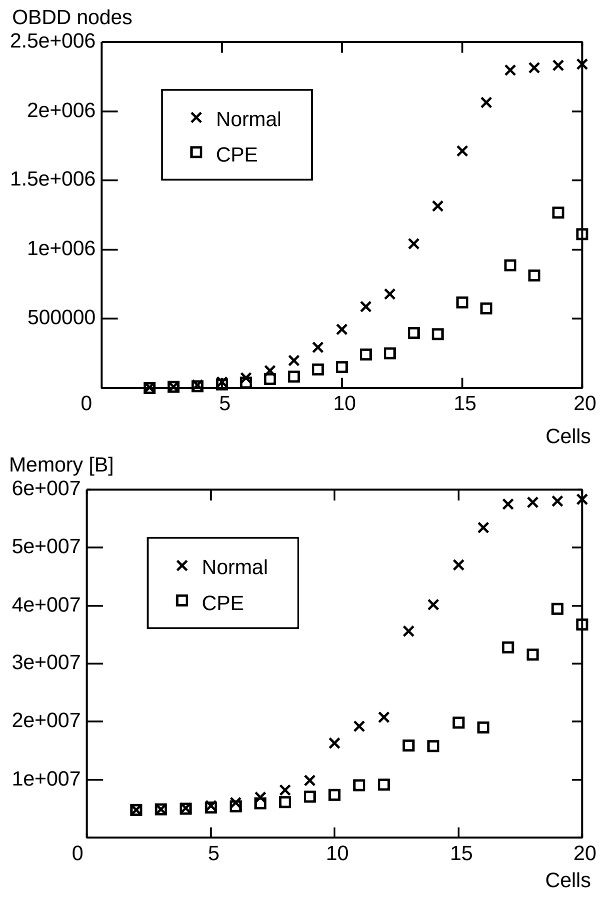


Figure 10: Verification performance of `mutex` by NuSMV: OBDD nodes(upper), amount of Memories(lower).

[3] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie: *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer, 2001.

[4] Kenneth L. McMillan: *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[5] Dov M. Gabbay, Mark A. Reynolds, and Marcelo Finger: *Temporal Logic Mathematical Foundations and Computational Aspects*, Volume 2, Oxford Science Publications, 2000.

[6] Sung-Tae Jung and Chris J. Myers: ”Direct Synthesis of Timed Circuits From Free-Choice STGs,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.21, No.3, pp.275–290, March 2002.

[7] Alex Yakovlev, Luis Gomes and Luciano Lavagno: *Hardware Design and Petri Nets*, Kluwer Academic Publishers, 2000.

[8] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A.Yakovlev: ”Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers,” *IEICE Transactions on Information and Systems*, Vol.E80–D, No.3, pp.315–325, 1997.

[9] Gerard J. Holzmann: *The SPIN Model Checker – Primer and Reference Manual*, Addison-Wesley, 2004.

[10] `http://spinroot.com/`

[11] `http://nusmv.irst.itc.it/`