Vectorized Model Checking for Computation Tree Logic

Hiromi Hiraishi, Shintaro Meki and Kiyoharu Hamaguchi
Department of Information Science, Kyoto University
Sakyo-ku, Kyoto, 606, Japan.

Abstract

The aim of this paper is to show how big model checking problems for Computation Tree Logic (CTL) can be handled by using current powerful vector processors. Although efficient recursive model checking algorithms for CTL, which run in time proportional to both the size of Kripke structures and the length of formulas, have been already proposed [7, 2], their algorithms cannot be vectorized due to recursive procedure calls. In this paper we propose a new model checking algorithm, called a vectorized model checking algorithm, for CTL which is suitable for the execution on vector processors. It can handle more than 1 million state Kripke structure derived from a deterministic sequential machine.

1 Introduction

Recently various kinds of formal methods for automatic verification have been widely studied. Among them, the model checking approach based on a branching time temporal logic called CTL (Computation Tree Logic) [2, 3, 5, 6, 7] is one of the most efficient approaches. In verification of a system which consists of several machines communicating with each other, however, there is a problem so-called a state explosion problem. Although there are several works trying to avoid this problem [8, 10], it seems to be difficult to avoid the problem in general, and there are strong requirements for verification of large systems[11].

We mainly aim at clarifying how big machines can be verified based on the model checking algorithm for CTL by using current powerful vector processors. As the first step to this purpose, we challenged to vectorize model checking for CTL on Kripke structures this time. Although the model checking algorithms in [7] are efficient and runs in time proportional to both the size of Kripke structures and the length of CTL formulas, they are not suitable for vector processors because it is difficult to vectorize them due to their recursive procedure calls. In order to extract high performance of vector processors, we need an algorithm using repeated uniform operations on array type data. It is easy to develop such a model checking algorithms based on fixpoint calculations of CTL semantics, but the direct implementation of the fixpoint calculations would easily lead to an algorithm whose time complexity is proportional to $|S|^3$ or $|S|^4$, where $|S|$ is the number of states of Kripke structure.

The new model checking algorithm called vectorized model checking algorithm proposed here can be vectorized for executions on vector processors. It runs in time linear to both the size of Kripke structures (i.e. $|S| + |R|$, where $|R|$ is the number of edges of Kripke structure) and the length of CTL formulas. We also implemented the algorithm on a vector processor FACOM VP400E. The analysis of storage requirement of the implementation shows that it can manipulate more than 1 million state Kripke structure derived from a deterministic sequential machine. We also present the result of an experiment which shows the efficiency of our implementation.

This paper is organized as follows: Section 2 summarizes the definition of CTL. Section 3 describes the vectorized model checking algorithm for CTL in conjunction with explanations about vector processors. In Section 4 we explain the implementation of the algorithm on a vector processor FACOM VP400E and show its experimental result. Section 5 concludes this paper with summarizing remaining future problems.

2 Computation Tree Logic

Computation Tree Logic (CTL)[6] is a branching time temporal logic. Let AP be a set of atomic propositions. CTL formulas are inductively defined as follows:

- If $p \in AP$, p is a CTL formula.
- If η is a CTL formula, then so are $\neg\eta$, $EX\eta$ and $EG\eta$.
- If η and ξ are CTL formulas, then so are $\eta \vee \xi$ and $E[\eta \mathcal{U} \xi]$.

The semantics of CTL is defined over a Kripke structure $K = (S, R, I)$, where

- S is a non-empty finite set of states.
- $R \subseteq S \times S$ is a total binary relation on S (i.e. for $\forall s \in S$, there exists $s' \in S$ such that $(s, s') \in R$).
- $I : S \rightarrow 2^{AP}$ is an interpretation function which labels each state with a set of atomic propositions true at that state.

An infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ is called a *path* from s_0 if $(s_i, s_{i+1}) \in R$ for $\forall i \geq 0$. $\pi(i)$ denotes the $i-$th state of the sequence π (i.e. $\pi(i) = s_i$).

The truth-value of a CTL formula is defined at a state of a Kripke structure and $K, s \models \eta$ denotes that a CTL formula η hold at a state s of a Kripke structure K. If there is no ambiguity, we will omit K and just write as $s \models \eta$. The relation $\models$ is recursively defined as follows:

- $s \models p$ $(\in AP)$ iff $p \in I(s)$.
- $s \models \neg\eta$ iff $s \not\models \eta$.
- $s \models \eta \vee \xi$ iff $s \models \eta$ or $s \models \xi$.
- $s \models EX\eta$ iff there exists some next state s' of s (i.e. $(s, s') \in R$) such that $s' \models \eta$.
- $s \models EG\eta$ iff there exists some path π on K starting from the state s such that $\pi(i) \models \eta$ for $\forall i \geq 0$.
- $s \models E[\eta \mathcal{U} \xi]$ iff there exists some path π on K starting from the state s such that $\exists i \geq 0$, $\pi(i) \models \xi$ and $\pi(j) \models \eta$ for $0 \leq \forall j < i$.

```
      DO 10 I = 1, N
   10 A(I) = B(I) + C(I)
```
(a) contiguous access

```
      DO 10 I = 1, N, K
   10 A(I) = B(I) + C(I)
```
(b) constant strided access

```
      DO 10 I = 1, N
   10 A(I) = B(L(I))
```
(c) indirectly addressed access

Figure 1: Three types of vector accesses.

```
      DO 10 I = 1, N
   10 IF (A(I) .GT. 0.0) B(I) = B(I) + C(I)
```
(a) DO loop with a conditional statement

```
      K = 0
      DO 10 I = 1, N
        IF (A(I) .GT. 0.0) THEN
          K = K + 1
          C(K) = B(I)
        ENDIF
   10 CONTINUE
```
(b) vector compress function

Figure 2: DO loops containing conditional statements

3 Vectorized Model Checking Algorithm

3.1 Vector processors

Vector processors are supercomputers for large-scale computations. They achieve more than several hundred MFLOPS (Million FLoating-point Operations Per Seconds) by vector instructions which execute uniform operations on array-structured data using pipelined functional units, and they usually have large main memory of several hundred mega bytes. In conjunction with floating-point operations, they also support integer and bit-wise logical operations.

Although the maximum speed of vector processors are very high, the following two points are very important in programming for vector processors to achieve their maximum performance:

Vectorization ratio: Vectorization ratio is the rate of the operations executed by vector instructions to the whole operations in a program. This ratio should be more than 90% to obtain high performance of vector processors.

Vector length: Since there are some overheads for setting up vector instructions, the length of operands (vector length) of vector instructions should be large enough; it should be larger than several hundreds to get maximum performance of vector processors.

As for data transmission between the main memory and vector registers, there are load/store pipelines which support basically the following three types of vector accesses: *contiguous vector access*, *constant strided vector access* and *indirectly addressed vector access* (see Figure 1).

Furthermore, vector processors support **DO** loop with conditional statements and *vector compress function* shown in Figure 2. These vectorized functions are very powerful in implementing vectorized model checker for CTL.

```
1. procedure Verify_Not(¬η)
2.  for all s ∈ S do
3.   Label(¬η, s) := Not(Label(η, s));
4.  return;
5. end of procedure
```

```
1. procedure Verify_Or(η ∨ ξ)
2.  for all s ∈ S do
3.   Label(η ∨ ξ, s) := Or(Label(η, s), Label(ξ, s));
4.  return;
5. end of procedure
```

Figure 3: Vecorized model checking algorithms for Boolean operators

3.2 Vectorized model checking algorithm

The vectorized model checking algorithm for CTL runs in bottom-up way by labeling each state with the truth value of each sub-formula contained in a given CTL formula. $Label(\eta, s)$ denotes the truth value of a CTL formula η at a state s in the following.

Case Boolean operations ($\neg\eta$ or $\eta \vee \xi$): We need only to calculate logical negation or disjunction of $Label(\eta, s)$ and $Label(\xi, s)$ for each state s as $Verify\_Not(\neg\eta)$ and $Verify\_Or(\eta \vee \xi)$ in Figure 3.

Case $EX\eta$: First, it assumes that $EX\eta$ is $false$ at all the states. For all states where η holds, it labels $EX\eta$ to be $true$ at their all predecessor states (see $Verify\_EX(EX\eta)$ in Figure 4.).

Case $E[\eta\,\mathcal{U}\,\xi]$: From the definition of $E[\eta\,\mathcal{U}\,\xi]$, it holds at the states where ξ is $true$ and it also holds at the states which are reachable to such states only through the states where η holds. Therefore, this is a kind of reachability problem.

In the procedure $Verify\_EU(E[\eta\,\mathcal{U}\,\xi])$, the sets of states N_1 and N_2 are used to keep track of the states where the truth value of $E[\eta\,\mathcal{U}\,\xi]$ is newly determined to be $true$. It first initializes the set N_1 and then it calculates the initial values of $Label(E[\eta\,\mathcal{U}\,\xi], s)$ at each state (lines $3 \sim 11$) as follows: if ξ holds at the state, it is 1 (i.e. $true$) because $E[\eta\,\mathcal{U}\,\xi]$ holds at the state apparently and the state is inserted to N_1; if neither ξ nor η hold at the state, it is 0 (i.e. $false$) because $E[\eta\,\mathcal{U}\,\xi]$ does not hold at the state apparently; if η holds but ξ does not, it is temporarily assigned to 2 indicating that it will be determined later by checking the reachability.

Next, for all the states labeled 2, the procedure checks the reachability to the state labeled 1 through the states labeled 2, and the reachable states are labeled 1 and the unreachable states are labeled 0. This step is done as follows (lines $12 \sim 27$): For each state where $E[\eta\,\mathcal{U}\,\xi]$ is newly determined to be $true$, the labels of its predecessor states are checked, and if they are 2, then they are relabeled as 1 because they are the reachable states and they are added to the set which keeps track of the states newly labeled as 1. This step is repeated until no more states are newly labeled as 1 (lines $12 \sim 23$). Finally, the states whose labels are still 2 are labeled as 0 because they are the unreachable states (lines $24 \sim 27$).

Case $EG\eta$: From the definition of $EG\eta$, it holds at the states on the loops which are constructed only by the states where η holds. It also holds at the states reachable to such loops through the states where η holds. In order to find out such states, the procedure $Verify\_EG((\eta))$ (Figure 4) first constructs a sub-graph logically by extracting all the states where η holds. Then it keeps removing the states from the sub-graph whose out degrees are 0 while such states exist. It is almost clear that $EG\eta$ holds at the states contained in

```
1.  procedure  Verify_EX(EXη)
2.   for all  s ∈ S do
3.    Label(EXη, s) := 0;
4.   for all  s' ∈ S do
5.    if  Label(η, s') = 1 then
6.     for all  s such that (s, s') ∈ R do
7.      Label(EXη, s) := 1;
8.   return;
9.  end of procedure
```

```
1.  procedure  Verify_EU(E[η U ξ])
2.   N1 := ∅
3.   for all  s ∈ S do
4.    if  Label(ξ, s) = 1 then
5.     begin
6.      Label(E[η U ξ], s) := 1;
7.      N1 := N1 ∪ {s};
8.     end
9.    else if  Label(η, s) = 0 then
10.    Label(E[η U ξ], s) := 0;
11.   else  Label(E[η U ξ], s) := 2;
12.  while  N1 ≠ ∅ do
13.   begin
14.    N2 := ∅;
15.    for all  s' ∈ N1 do
16.     for all  s such that (s, s') ∈ R do
17.      if  Label(E[η U ξ], s) = 2 then
18.       begin
19.        Label(E[η U ξ], s) := 1;
20.        N2 := N2 ∪ {s};
21.       end
22.    N1 := N2;
23.   end
24.  for all  s ∈ S
25.   if  Label(E[η U ξ], s) = 2 then
26.    Label(E[η U ξ], s) := 0;
27.  return;
28. end of procedure
```

```
1.  procedure  Verify_EG(EGη)
2.   N1 := ∅;
3.   for all  s ∈ S do
4.    if  Label(η, s) = 1 then
5.     begin
6.      Label(EGη, s) := 1;
7.      N1 := N1 ∪ {s};
8.     end
9.    else
10.    Label(EGη, s) := 0;
11.  if  N1 ≠ ∅ then
12.   begin
13.    for all  s' ∈ N1 do
14.     for all  s such that (s, s') ∈ R do
15.      if  Label(EGη, s) ≥ 1 then
16.       Label(EGη, s) := Label(EGη, s) + 1;
17.    N2 := ∅;
18.    for all  s ∈ N1 do
19.     if  Label(EGη, s) = 1 then
20.      begin
21.       Lable(EGη, s) := 0;
22.       N2 := N2 ∪ {s}
23.      end
24.    N1 := N2;
25.   end
26.  while  N1 ≠ ∅ do
27.   begin
28.    N2 := ∅;
29.    for all  s' ∈ N1 do
30.     for all  s such that (s, s') ∈ R do
31.      begin
32.       if  Label(EGη, s) = 2 then
33.        N1 := N1 ∪ {s};
34.       Label(EGη, s) := Label(EGη, s) − 1;
35.      end
36.    N1 := N2;
37.   end
38.  for all  s ∈ S do
39.   if  Label(EGη, s) ≥ 2 then
40.    Label(EGη, s) := 1;
41.   else
42.    Label(EGη, s) := 0;
43.  return;
44. end of procedure
```

Figure 4: Vectorized model checking algorithms for temporal operators

the resulting sub-graph and it does not hold at all other states.

More precisely, after initializing the set of states N_1 to be empty, the procedure labels the states as 1 where η holds; it labels the states as 0 where η does not hold (lines 3 $\sim$ 10). For each state labeled 1, if the labels of its predecessor states are greater than 0, then they are incremented (lines 13 $\sim$ 16). At this point, the label 0 means that $EG\eta$ does not hold at the state; the label greater than 0 means that η holds at the state and it has its label - 1 successor states where η holds. Next, for each state labeled 1 (i.e. the state which has no successor states where η holds), the label is relabeled to 0 and the state is inserted to the set N_2 which keeps track of the states newly labeled as 0 (lines 18 $\sim$ 23). In lines 27 $\sim$ 37, for each state newly determined that $EG\eta$ does not hold on it, the labels of its predecessor states are decremented. This step is repeated for those states that become to have no successor states until no more such states exist. Finally, the states whose labels are greater than 1 are relabeled to 1 because such states have at least one infinite path on which η always holds; the other states are labeled as 0 (lines 38 $\sim$ 43).

3.3 Time complexity

It is clear that *Verify_Not*($\neg\eta$) and *Verify_Or*($\eta \vee \xi$) runs in time proportional to $|S|$.

In the case of *Verify_EX*($EX\eta$), the lines 6 $\sim$ 7 are executed only $|R|$ times in total by adopting a data structure which assigns a list of its predecessor states to each state. Therefore, the time complexity is $O(|S| + |R|)$.

In the case of *Verify_EU*($E[\eta\, \mathcal{U}\, \xi]$), the *for loop* from the line 3 to the line 11 is executed $|S|$ times. The lines 16 $\sim$ 21 are executed only $|R|$ times in total because it never checks the predecessor states of the same state twice. The last *for loop* (lines 24 $\sim$ 26) is repeated $|S|$ times. Therefore, its time complexity is $O(|S| + |R|)$.

In the case of *Verify_EG*($EG\eta$), it is easy to see that the lines 3 $\sim$ 10, 18 $\sim$ 23, and 38 $\sim$ 43 are executed $|S|$ times in total, and the lines 14 $\sim$ 16 and 30 $\sim$ 35 are repeated $|R|$ times in total in the same way as in the case of *Verify_EU*($E[\eta\, \mathcal{U}\, \xi]$). Therefore, its time complexity is $O(|S| + |R|)$.

Since one of these procedures is executed for each sub-formula of a given CTL formula η, the time complexity of the vectorized model checking algorithm is $O((|S| + |R|)|\eta|)$, where $|\eta|$ denotes the length of η.

3.4 Fairness constraints

Fairness constraints can be handled efficiently by labeling *fair states* which have at least one *fair path* [7, 9]. This can be done by first obtaining *fair strongly connected components* and then getting reachable states to the fair strongly connected components.

There is a well known linear time algorithm to get strongly connected components of a directed graph based on the depth first search [1]. It seems to be difficult to vectorize this algorithm. We leave the vectorization of this part as a future problem and decided to use the non-vectorized well known algorithm.

Once the strongly connected components have been obtained, it is easy to vectorize the decision procedure if they are fair or not. The reachability problem can be also vectorized in the same way as model checking of $E[\eta\, \mathcal{U}\, \xi]$.

The labeling for fair states should be done once before starting model checking and it should be also done when evaluating EG operator.

4 Vectorized Model Checker

4.1 Implementation

We have implemented the vectorized model checking algorithm on a vector processor FACOM VP400E as a vectorized model checker. In the VP400E, three pipelined vector functional units, each of which consists of 4 pipelined units, can operate in parallel with 7 nano second cycle time. Its peak performance is about 1714 MFLOPS. It has a 256 M byte main memory, in which we can use 200 M bytes as a user area.

The input of the vectorized model checker is a Moore type deterministic sequential machine and CTL formulas to be verified. It creates the corresponding Kripke structure internally from a given Moore type deterministic sequential machine.

Let $M = (X, Z, \Sigma, \delta, \lambda, s_0)$ be a Moore type deterministic sequential machine, where

- X is a finite and nonempty set of binary input signals (atomic propositions);
- Z is a finite and nonempty set of binary output signals (atomic propositions);
- Σ is a finite and nonempty set of states;
- $\delta : 2^X \times \Sigma \rightarrow \Sigma$ is the state transition function;
- $\lambda : \Sigma \rightarrow 2^Z$ is the output function;
- s_0 is the initial state.

Then, the corresponding Kripke structure $K = (S, R, I)$ becomes as follows:

$$\begin{aligned} S &= \{s'_{i,j} | s_i \in \Sigma, j \in 2^X \text{and } \delta(j, s_i) \text{ is defined.}\} \\ R &= \{(s_{i,j}, s_{i',j'}) | s_{i,j}, s_{i',j'} \in S, \delta(j, s_i) = s_{i'}\} \\ I(s'_{i,j}) &= \{x | x \in j\} \cup \{z | z \in \lambda(s_i)\} \end{aligned}$$

Intuitively, there is a one to one correspondence between the transition edges of the Moore type deterministic sequential machine M and the states of the corresponding Kripke structure K. The size of the Kripke structure becomes as follows:

$$\begin{aligned} O(|S|) &= O(|\Sigma| \times 2^{|X|}) \\ O(|R|) &= O(|E| \times 2^{|X|} \\ O(|S| + |R|) &= O((|\Sigma| + |E|) \times 2^{|X|}), \end{aligned}$$

where $|E|$ represents the number of transition edges of M and $O(|E|) = O(|\Sigma| \times 2^{|X|})$.

Almost all parts of the model checker are vectorized except the fairness constraints handling. The most time consuming parts of the vectorized model checker are the calculations of labels of the predecessor states for each state (lines 16 $\sim$ 21 of *Verify_EU*$(E[\eta \mathcal{U} \xi])$ and lines 30 $\sim$ 35 of *Verify_EG*$(EG\eta)$ in Figure 4). The average vector length for these parts becomes the average number of predecessor states of each state, and it is $|R|/|S|$ or $2^{|X|}$. That is, if the machine has 10 input signals, the average vector length becomes 1024 and it is enough large to extract high performance of a vector processor.

In order to represent a transition relation R of a Kripke structure, we use 2 integer arrays Q and R. Since there is a one to one correspondence between the edges of a sequential machine and the states of the corresponding Kripke structure, it is possible to number the states of the Kripke structure so that each state s_i has its predecessor states $s_{Q(i)} \sim s_{R(i)}$. By using this numbering method, we can use the efficient *contiguous*

Table 1: Kripke structures for benchmark tests

Name	$\|Spec\|$		#States	#Edges
SR8	Total:	306	131,072	67,108,864
	Boolean:	212		
	Temporal:	94		
SR9	Total:	368	524,288	536,870,912
	Boolean:	254		
	Temporal:	114		

access (see Figure 1) to calculate the labels of predecessor states which is the most time consuming parts of the model checker. The sizes of the arrays Q and R are both $|S|$. These 2 integer arrays Q and R can be easily constructed directly from a given Moore type deterministic sequential machine in proportional time to the size of the corresponding Kripke structures by using one additional integer array of size $|S|$.

As for N_1 and N_2 used in *Verify_EU*$(E[\eta \mathcal{U} \xi])$ and *Verify_EG*$(EG\eta)$, we also use integer arrays with the corresponding index variables. Initialization of N_i (i.e. $N_i := \emptyset, i = 1, 2$) can be done by just substituting 0 to the corresponding index variable. Insertion of a state s to N_i (i.e. $N_i := N_i \cup \{s\}$) can be done by just storing the state s at the place in N_i pointed by its corresponding index variable and updating the index variable. As for copying data from N_2 to N_1 (i.e. $N_1 := N_2$), we just exchange the role of N_1 and N_2 instead of copying data actually. The maximum required sizes of the arrays N_1 and N_2 are $|S|$.

In order to store the truth value of each sub-formula at each state, we use 1 bit each. Therefore, the required memory in total for this purpose is $|S| \times |\eta|/8$ bytes. In addition, we use a working integer array of size $|S|$ to store a label for each state in checking temporal operators.

We also use $|S| \times 8$ words to handle fairness constraints.

Note that 1 integer word consists of 4 bytes. Therefore, the total amount of required memory is

$$|S| \times 56 + |S| \times |\eta|/8 \text{ bytes.}$$

For CTL formulas which contains 256 and 1024 operators, the vectorized model checker can manipulate Kripke structures of 2.3 million and 1.1 million states respectively with main memory of 200 M bytes.

4.2 Example

In order to measure the efficiency of the vectorized model checker, we applied it to the verification of two large Kripke structures (SR8 and SR9) corresponding to synchronous shift registers with parallel load and serial output. SR8 consists of 131,072 states and 67,108,864 edges (each state has 512 edges), and SR9 consists of 524,288 states and 536,870,912 edges (each state has 1024 edges) as shown in Table 1. The CTL formulas which give their full specifications contain more than 300 different sub-formulas with no fairness constraints.

The benchmark results without fairness handling are shown in Table 2. Both SR8 and SR9 are verified to be *true* on the vector machine of VP-400E in about 5 seconds

Table 2: Results of benchmark tests

Name	Scalar Execution (m sec.)	Vecotr Execution (m sec.)	Acceleration Ratio	Average Vector Length	Used Memory
SR8	132,126	4,998	26.4	512	13 MB
SR9	1,131,039	28,985	39.0	1,024	52 MB

and 29 seconds by using 13 MB and 52 MB of memory respectively. This means that our vectorized model checker evaluates about $7 \sim 8$ states in a second, which implies that it will be able to verify 1 million state Kripke structure in a couple of minutes. Furthermore, the acceleration ratio obtained by our algorithm is around $26 \sim 39$, which is extremely high ratio in non numeric application programs.

We also verified SR8 by using the CTL model checker B1.0 developed by Clarke et al. [5] installed on Sun-3/80. It took about 5,068 seconds to verify SR8. By considering that some benchmark tests show that the scalar unit of FACOM VP-400E is about 6.7 times faster than Sun-3/80, it will still take about 750 seconds to verify SR8 even if we install the CTL model checker B1.0 on VP-400E because their algorithm cannot be vectorized due to recursive calls.

5 Conclusion

We proposed a vectorized model checking algorithm for CTL and implemented the vectorized model checker on a vector processor FACOM VP400E. Almost all parts of the algorithms are vectorized except the parts to obtain strongly connected components for fairness constraints handling.

It can handle about 2.3 million or 1.1 million state Kripke structures derived from a Moore type deterministic sequential machine when the length of a give CTL formula is less than 256 or 1024 respectively.

We also presented examples which show that a CTL formula of 368 different subformulas is verified to be true in 29 seconds on a Kripke structure with 524,288 states and 536,870,912 edges.

There are several remaining future problems:

The first is to devise a vectorized algorithm for obtaining strongly connected components of a directed graph, and we need more consideration.

The second is a vectorization of model checking which can handle sequential machines directly [2]. We think this is not so difficult and would like to implement it in the near future.

The third is a vectorization of model checking based on Binary Decision Diagrams (BDD) because the BDD is a powerful technique to reduce necessary amount of memory for model checking dramatically in some cases [4].

Although our current version of the vectorized model checker takes a single Moore type sequential machine as input, it would be interesting to challenge the vectorization of the procedure to create direct product of several concurrent/parallel sequential machines. It would be also interesting to devise vectorized algorithm for model checking without creating direct product explicitly.

Acknowledgments The authors would like to express their appreciations to Prof. E. M. Clarke of CMU for his valuable suggestions. They also would like to express their appreciations to Prof. S. Yajima, Dr. N. Takagi, Mr. N. Ishiura and Mr. H. Ochi of Kyoto University for their precious discussions and comments.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, 1974.

[2] M. C. Browne. An improved algorithm for the automatic verification of finite state systems using temporal logic. Technical Report CMU-CS-86-156, Carnegie Mellon University, 1986.

[3] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. Logic in Computer Science*, 1990.

[5] E. M. Clarke, S. Bose, M. C. Browne, and O. Grumberg. The design and verification of finite state hardware controllers. Technical Report CMU-CS-87-145, Carnegie Mellon University, July 1987.

[6] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Proc. Workshop on Logic of Programs*, pages 52–71. Springer-Verlag, 1981.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. Technical Report CMU-CS-83-152, Carnegie Mellon University, 1983.

[8] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking algorithms. In *Proc. 6th Annual ACM Symposium of Principles of Distributed Computing*, pages 294–303, August 1987.

[9] E. M. Clarke and O. Grumberg. Research on automatic verification of finite-state concurrent systems. Technical Report CMU-CS-87-105, Carnegie Mellon University, January 1987.

[10] E. M. Clarke, D. E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. In *Proc. 9th IFIP Symposium on Computer Hardware Description Languages and their Applications*, pages 281–295, June 1989.

[11] S. Graf, J. L. Richier, C. Rodriguez, and J. Voiron. What are the limits of model checking methods for the verification of real life protocols ? In *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, June 1989.