Symbolic Model Checking for Self-Stabilizing Algorithms

Tatsuhiro Tsuchiya, *Member*, *IEEE*, Shin'ichi Nagano, *Member*, *IEEE*, Rohayu Bt Paidi, *Member*, *IEEE*, and Tohru Kikuno, *Member*, *IEEE*

Abstract—A distributed system is said to be self-stabilizing if it converges to safe states regardless of its initial state. In this paper we present our results of using symbolic model checking to verify distributed algorithms against the self-stabilizing property. In general, the most difficult problem with model checking is state explosion; it is especially serious in verifying the self-stabilizing property, since it requires the examination of all possible initial states. So far applying model checking to self-stabilizing algorithms has not been successful due to the problem of state explosion. In order to overcome this difficulty, we propose to use symbolic model checking for this purpose. Symbolic model checking is a verification method which uses Ordered Binary Decision Diagrams (OBDDs) to compactly represent state spaces. Unlike other model checking techniques, this method has the advantage that most of its computations do not depend on the initial states. We show how to verify the correctness of algorithms by means of SMV, a well-known symbolic model checker. By applying the proposed approach to several algorithms in the literature, we demonstrate empirically that the state spaces of self-stabilizing algorithms can be represented by OBDDs very efficiently. Through these case studies, we also demonstrate the usefulness of the proposed approach in detecting errors.

Index Terms—Self-stabilization, automatic verification, symbolic model checking, distributed algorithms.

1 Introduction

A DISTRIBUTED system is said to be *self-stabilizing* if it satisfies the following two properties: 1) Convergence—the system reaches a safe state regardless of its initial state, and 2) closure—once the system reaches a safe state, it continues to be within the set of safe states. The idea of self-stabilization was first introduced to computer science by Dijkstra [5]. This idea, which originally had a very narrow scope of application, has attracted much research interest in recent years (cf. [22]). In general, a self-stabilizing system has two useful properties: 1) It need not be initialized, and 2) it can recover from transient faults that may change its state. These properties are very useful in distributed environments where no centralized control exists.

In this paper, we discuss automatic verification of self-stabilizing algorithms. Automatic verification is relatively unexplored in the field of self-stabilizing algorithms, due to its awkwardness.

There are two distinct traditions in automatic verification. One is *mechanical theorem proving*, and the other is *model checking*. The first approach has been discussed by several researchers in the context of self-stabilizing algorithms. In [19], Prasetya verified a self-stabilizing minimum-cost routing algorithm using the HOL proof checking system [9]. In [20], Qadeer and Shankar applied PVS [17] to prove the correctness of Dijkstra's self-stabilizing ring algorithm [5]. Recently, Kulkarni et al. [15] also proved the correctness of the Dijkstra's algorithm using PVS in a different fashion. Generally, mechanical theorem proving is a highly powerful and flexible approach. For example, it can be used for reasoning about infinite state systems. Unfortunately, this approach can involve generating and proving many lemmas to verify the correctness of systems. Although this process can be automated to some extent by means of proof checking systems, proofs must still be constructed mainly by hand. Consequently, mechanical theorem proving can be performed only by experts who have considerable experience in logical reasoning.

The second approach to automatic verification, the model checking, is the process of exploring a finite state space to determine whether or not a given property holds. This is often the easiest way to verify distributed algorithms; however, it is more limited. This leads to disadvantages, such as only being able to apply it to finite state systems, and it is impractical when the state space is very large, even though it is finite. The latter problem, which often occurs when the system being verified has many components, is usually referred to as the *state explosion problem*.

At the same time, model checking has two remarkable advantages; first, it is fully automatic and its application does not require the user to have mathematical knowledge such as theorem proving. Second, when the design fails to satisfy a desired property, the process of model checking produces a counterexample that demonstrates a behavior which invalidates the property. Therefore, the use of model checking can be useful for algorithm designers who need to

- *T. Tsuchiya and T. Kikuno are with the Department of Informatics and Mathematical Science, Osaka University, Osaka 560-8531, Japan. E-mail: {t-tutiya, kikuno}@ics.es.osaka-u.ac.jp.*
- *S. Nagano is with Toshiba Corporate Research and Development Center, Kawasaki 212-8582, Japan. E-mail: shinichi3.nagano@toshiba.co.jp.*
- *R.B. Paidi is with Matsushita Electric Company, Malaysia. E-mail: rohayu@po3.ved.mei.co.jp.*

Manuscript received 12 Aug. 1999; revised 15 Aug. 2000; accepted 15 Sept. 2000.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 110427.

validate distributed algorithms, especially in early stages of development.

However, since the state explosion problem is especially serious in verifying a self-stabilizing algorithm, applying model checking to them has not been successful so far (note that since any state can be the initial state, the set of the reachable states is exactly the same as the Cartesian product of sets of states of all components). As far as we know, the only work that reports the results of using model checking for verifying self-stabilization is the one by Shukla et al. [23]. For two distributed algorithms, they verified whether the system converges to safe states from a given initial state, using a software tool called SPIN [12]. Nevertheless, their method cannot be directly used for verifying whether this property holds for all possible initial states. This problem can be alleviated by minor modifications that allow any state to be an initial state. As shown later, however, these modifications render the method infeasible even for small systems.

To overcome the problem of applying model checking to self-stabilizing algorithms, we propose to use *CTL symbolic model checking* (symbolic model checking, for short). This method controls the state explosion problem by using Boolean functions to implicitly represent the state space. Since Boolean functions can be often represented by *Ordered Binary Decision Diagrams* (OBDDs) very compactly, the symbolic model checking method can reduce the memory and time required for analysis. By manipulating the Boolean functions, the method can determine whether a system meets a given property that is specified using CTL [3], a branching time temporal logic.

Compared to other model checking approaches, the symbolic model checking method has several features that are appropriate for verifying self-stabilizing algorithms. First, since most of the computations required by the method do not depend on reachability of states, the property that any state can be the initial state never becomes an obstacle to verification. Second, the self-stabilizing property can be expressed by a simple CTL formula. Third, as will be empirically shown later, the state spaces of self-stabilizing systems can be represented very compactly by using OBDDs. Besides, a symbolic model checking tool called SMV (Symbolic Model Verifier) [16] is widely available.

In this paper, we investigate how we can verify algorithms against the self-stabilizing property by using SMV. To illustrate the feasibility of our approach, we describe the results of applying it to several algorithms proposed in the literature. During the verification process, we found an error in one of these algorithms [26].

The remainder of this paper is organized as follows: In the next section, we describe the concept of self-stabilizing algorithms; in Section 3, we briefly explain symbolic model checking and the symbolic model checker SMV. In Section 4, we present how to verify distributed algorithms against the self-stabilizing property by using SMV. By applying the approach to several algorithms, we demonstrate its applicability in Section 5. For comparative purposes, we show the results of using SPIN, a model checker based on explicit state enumeration, for validation of a self-stabilizing algorithm in Section 6; and we conclude our paper with a brief summary in Section 7.

2 Self-Stabilizing Algorithms

2.1 Models and Definitions

We consider a distributed system that consists of n processes, $p_0, p_1, p_2, \cdots, p_{n-1}$. For convenience, the subscripts on p_i are assumed to be modulo n. The topology of the system is modeled by an undirected graph of which each vertex corresponds to a process. Process p_i can communicate with another process, p_j, if p_i and p_j are adjacent to each other on the graph.

We consider two models of communication: In the *state-reading model*, each process can directly read the internal state of its neighboring processes; in the *link-register model*, processes can communicate with each other only by using separate registers. In the latter model, there are two registers R_{ij} and R_{ji} for each adjacent pair of processes p_i and p_j. Process p_j can read the state of R_{ij} but not the state of p_i itself, and only p_i can change the state of R_{ij}. We call R_{ij} and R_{ji} the *output register* and the *input register* of p_i for p_j, respectively. Thus, R_{ij} is the input register of p_j for p_i. The number of registers is denoted by l.

We assume that the number of the states of each component (process or register) of the system is finite and we define the *global state* of the system as the vector of the states of all components. Therefore, the set of all global states, denoted by $\mathcal{G}$, is given as follows:

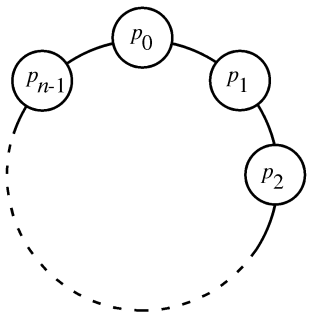
- the state-reading model $\mathcal{G} = Q_0 \times Q_1 \times \cdots \times Q_{n-1}$, and
- the link-register model

$$\mathcal{G} = Q_0 \times Q_1 \times \cdots \times Q_{n-1} \times O_0 \times O_1 \times \cdots \times O_{l-1},$$

where $Q_i (0 \le i \le n-1)$ and $O_i (0 \le i \le l-1)$ denote the set of states of p_i and the set of states of the $(i+1)$th register, respectively.

A distributed algorithm specifies a transition relation for each process p_i. Based on the transition relation, p_i reads the states of its neighboring processes or its input registers, calculates the next local state, and updates, if needed, its output registers in each step of execution. A distributed algorithm thus specifies the behavior of the system, and in this paper, we limit our discussion to deterministic algorithms.

Concerning selection of processes to run, two types of daemons are considered: the *central daemon* (c-daemon) and the *distributed daemon* (d-daemon). If the c-daemon is assumed, then only one process is selected to run at a time, while an arbitrary set of processes is selected to run under the d-daemon. For either type of daemon, we assume it to be *fair*, that is, we assume that each process is selected infinitely often. We use $g \xrightarrow{U} g'$ by express the fact that processes in $U (\subseteq \{p_0, p_1, \cdots, p_{n-1}\})$ are selected at $g \in \mathcal{G}$ and yield $g' \in \mathcal{G}$ by their parallel execution. (If U is not important or is clear, we may omit it.) An infinite sequence of global states $g_0 g_1 g_2 \cdots$ is called a *computation* iff for every $i (\ge 0)$ there is $U_i (\subseteq \{p_0, p_1, \cdots, p_{n-1}\})$ such that $g_i \xrightarrow{U_i} g_{i+1}$. A



(a)

p_0	p_1	p_2	privileged processes	process to make move
2	1	2	p_0, p_1, p_2	p_0
3	1	2	p_1, p_2	p_1
3	3	2	p_2	p_2
3	3	3	p_0	p_0
0	3	3	p_1	p_1
0	0	3	p_2	p_2
0	0	0	p_0	p_0
1	0	0	p_1	—

(b)

Fig. 1. (a) A ring network. (b) An example of a computation of the *K*-state algorithm ($n = 3$, $K = 4$).

computation is said to be *fair* if it is produced by a fair daemon.

Self-stabilization is defined as follows: Let $\mathcal{L}$ be the set of the *legitimate* (or safe) states in which the system performs correct execution. A distributed system is said to be *self-stabilizing* if it satisfies the following two properties:

1. Convergence—for any global state $g_0 \in \mathcal{G}$ and any fair computation $g_0 g_1 g_2 \cdots$ starting with g_0, there is an integer $k(\geq 0)$ such that $g_k \in \mathcal{L}$, and
2. Closure—for any global state $g \in \mathcal{L}$, $g \rightarrow g'$ implies $g' \in \mathcal{L}$.

An algorithm can be defined as self-stabilizing in a corresponding manner, thus, a self-stabilizing algorithm specifies a self-stabilizing system.

2.2 Illustrative Example

Here we take Dijkstra's *K*-state mutual exclusion algorithm as an illustrative example [5]. Consider a distributed system that consists of n processes connected in the form of a ring, as shown in Fig. 1a. We assume the state-reading model and the existence of the c-daemon, and we define a *privilege* of a process as its ability to change its current state. This ability is based on a Boolean predicate that consists of its current state and the state of one of its neighboring processes.

We then define the legitimate states as those in which the following two properties hold: 1) exactly one process has a privilege, and 2) every process will eventually have a privilege. These properties correspond to a form of mutual exclusion, because the privileged process can be regarded as the only process that is allowed in its critical section.

In the *K*-state algorithm, the state of each process is in $\{0, 1, 2, \cdots, K-1\}$, where K is an integer larger than or equal to n. For any process p_i, we use the symbols S and L to denote its state and the state of its neighbor p_{i-1}, respectively, and process p_0 is treated differently from all other processes. The *K*-state algorithm is described below.

- process p_0
  if $(L = S)\{S := (S + 1) \bmod K;\}$
- process $p_i (i = 1, 2, \cdots, n-1)$
  if $(L \neq S)\{S := L;\}$.

Fig. 1b shows part of a computation of the system with three processes and $K = 4$. Although every process has a privilege initially, after two steps the system reaches a state where only one process is privileged. Thereafter, there is exactly one privileged process in the system and each process has privilege infinitely often. The computation shown in Fig. 1b is only an example, since the running processes are selected arbitrarily by the daemon assumed. Nevertheless, one can prove that by using this algorithm the system converges such legitimate states regardless of the initial state and computation [6].

3 Symbolic Model Checking

Model checking is the process of exploring a finite state space to determine whether or not a given property holds. The major problem of model checking is that the state spaces arising from practical problems are often extremely large, generally making exhaustive exploration not feasible.

A promising approach to this problem is the use of symbolic representations of the state space. In *CTL symbolic model checking* (symbolic model checking for short), Boolean functions represented by Ordered Binary Decision Diagrams (OBDDs) are used to represent the state space, instead of explicit adjacency-lists. This can reduce dramatically the memory and time required because OBDDs represent many frequently occurring Boolean functions very compactly.

Consider a set of Boolean vectors $\mathcal{B} = \{\text{true}, \text{false}\}^c$. Then any subset of $\mathcal{B}$ can be represented by a Boolean function (say B) with c Boolean variables such that the vector $\boldsymbol{x} \in \mathcal{B}$ is in the subset if and only if $B(\boldsymbol{x})$ is true. Since $\mathcal{B} = \{\text{true}, \text{false}\}^c$ has 2^c elements, 2^c states can thus be handled by using c Boolean variables. The transition relation is also represented by a Boolean function F with $2c$ variables such that there is a transition from $\boldsymbol{x}$ to $\boldsymbol{y}$ if and only if $F(\boldsymbol{x}, \boldsymbol{y})$ is true ($\boldsymbol{x}, \boldsymbol{y} \in \mathcal{B}$). Since the Boolean function F can be defined without any information on reachability, it can be constructed regardless of the initial states.

The correctness property to be verified is specified in CTL (Computational Tree Logic) [3]. CTL is a branching-time temporal logic, extending propositional logic with temporal operators that express how propositions change their truth values over time. Here we only use three temporal operators: AG, AF, and AX. The formula AG p holds in state s if p holds in all states along all computation paths (i.e., sequences of states) starting from s, while the formula AF p holds in state s if p holds in some state along all computation paths starting from s. The formula

AX p holds in state s if p holds in all the states that can be reached from s in exactly one step.[1] An atomic proposition is a CTL formula. If f_1 and f_2 are CTL formulae, then so are $\neg f_1$, $f_1 \wedge f_2$, AF f_1, AG f_1, and AX f_1.

In the process of symbolic model checking, a given CTL formula is evaluated with respect to all the initial states as follows: First, the set of all states where the given property holds is computed from the transition relation function F. This is done by fixed-point iterative techniques which manipulate Boolean functions encoded as OBDDs. (See [2], [16] for details.) Finally, whether the set obtained contains all initial states is determined. If it contains all the initial states, then the system meets the correctness property.

Only the final phase of the model checking process is thus related to the initial states, and most of the computations required do not depend on the state space reachable from the initial states. Consequently, this characteristic can be a drawback of the symbolic model checking, since the states that are never reached must be explored. However, in the case of self-stabilizing systems, this property never becomes a factor that worsens the verification performance, because all states are necessarily reachable.

4 Verifying Self-Stabilization Using SMV

4.1 SMV

SMV (Symbolic Model Verifier) [16] is a software tool for symbolic model checking; it is publicly available and has been especially successful in verifying hardware systems. In this section, we describe how we can use SMV to verify self-stabilizing algorithms.

In SMV, a system (or an algorithm) to be verified is described in a special language called the *SMV language*. We refer to a system description written in the SMV language as an *SMV program*. An SMV program is divided into one or more modules, each of which specifies a finite state machine. Each module contains variable declarations to determine its state space and descriptions of the initial state and transition relation of the machine.

Variable declarations are preceded by the keyword `VAR`. The type associated with a variable can be Boolean or an enumerated type. The transition relation is described by a collection of parallel assignments to the next version of the variables. Assignments of initial values and next values to the variables are preceded by the keyword `ASSIGN`. Initial states are assigned by specifying the initial values of the variables using the expression `init(x)`, where `x` is a variable. The expression `next(x)` is used to refer to the variable `x` in the next state.

For example, consider a finite state machine that has three states, say $s1$, $s2$, and $s3$, and suppose that $s1$ is the initial state and that the state nondeterministically changes at every move. This machine is represented in the SMV language as follows:

```
MODULE p
VAR     state : {s1, s2, s3}
ASSIGN  init(state) := s1;
        next(state) := {s1, s2, s3};
```

For the details of the syntax and semantics of the SMV language, the readers are referred to [16].

1. Operators G, F, and X mean *globally*, *sometime in the future*, and *next time*, respectively. In CTL, these operators must be preceded by a path quantifier which is either A (*for all computation paths*) or E (*for some computation path*).

```
MODULE main
VAR     p0 : process type_p(p2.state);
        p1 : process type_q(p0.state);
        p2 : process type_q(p1.state);

DEFINE condition1 := (( p0.priv & !p1.priv & !p2.priv) |
                      (!p0.priv &  p1.priv & !p2.priv) |
                      (!p0.priv & !p1.priv &  p2.priv));
       condition2 := (AF p0.priv & AF p1.priv & AF p2.priv);
       legitimate :=  condition1 & condition2;

SPEC   AF legitimate  & (legitimate -> AX legitimate)

MODULE type_p(L)
VAR     state : {0, 1, 2, 3};
DEFINE  priv := (state = L);
ASSIGN  init(state) := {0, 1, 2, 3};
        next(state) :=  case
                          priv : (state + 1) mod 4;
                          1 : state;
                        esac;
FAIRNESS  running

MODULE type_q(L)
VAR     state : {0, 1, 2, 3};
DEFINE  priv := !(state = L);
ASSIGN  init(state) := {0, 1, 2, 3};
        next(state) := case
                         priv : L;
                         1 : state;
                       esac;
FAIRNESS    running
```

Fig. 2. An SMV program for the *K*-state algorithm ($n = 3, K = 4$).

4.2 Describing Algorithms in the SMV Language

Here we explain how to represent a distributed algorithm in the SMV language and how to verify it against the self-stabilizing property using SMV. For this purpose, we take the *K*-state algorithm as an example, assuming that the c-daemon exists. (The d-daemon will be discussed in the next subsection.) Fig. 2 shows the SMV program that represents the *K*-state algorithm where $n = 3$ and $K = 4$.

4.2.1 Interaction between Processes

The interaction between processes is specified in the `main` module. The SMV language allows modular hierarchical descriptions and definition of reusable components. The `main` module defines the interaction of other modules at a lower level, each of which represents the behavior of a process.

The `main` module in Fig. 2 declares three processes, p_0, p_1, and p_2. The behavior of p_0 and that of $p_i (i = 1, 2)$ are specified by modules `type_p` and `type_q`, respectively. The `main` module also specifies that each process $p_i (i = 0, 1, 2)$ can refer to the value of a variable `state` of another process p_{i-1}. This corresponds to the fact that the processes are connected in the form of a ring. (As described later, network topologies other than rings can be specified in a similar way.)

In the `main` module in this program, the three processes are associated with the keyword `process`. In SMV, such

instances of modules are not allowed to run simultaneously. That is, at every step, at most one of them is nondeterministically selected and allowed to update its own state. The SMV program in Fig. 2 thus represents the existence of the c-daemon, which arbitrarily selects one process at a time for running.

4.2.2 Processes

As stated above, the behavior of each process is expressed by a module in the SMV program. The next local state of the process is determined depending on its current local state and the local states of its adjacent processes or registers in the network.

In modules `type_p` and `type_q` in Fig. 2, variable `state` denotes the state of the corresponding process p_i, while `L` aliases variable `state` of its left neighbor p_{i-1}, that is, it denotes the state of the left neighbor.

The keyword `DEFINE` is used to associate a symbol with a commonly used expression. In `type_p` and `type_q`, it is used to assign expressions `(state = L)` and `!(state = L)` to symbol `priv`, respectively. Therefore, `priv` takes the truth value iff the corresponding process has a privilege. (`!` represents negation.)

The value of `next(state)`, i.e., the next state of the process changes depending on the values of `state` and `L` as follows. The value of a `case` expression is determined by the first expression on the right hand side of a ":" such that the condition on the left hand side is true. Thus, for process p_0, if `priv` is true, then the result of the expression is `(state + 1) mod 4`; otherwise, it is `state`, which means the value of `state` does not change. (`1` and `0` represent the truth value and the false value, respectively.)

The keyword `FAIRNESS` and a CTL formula force SMV to verify only computation paths where the CTL formula becomes true infinitely often. Each process has a special variable `running` which is true iff that process is currently being executed. Thus, by adding the declaration

```
FAIRNESS running
```

to each process, we can limit computation paths to be verified to those in which `running` of every process has the truth value infinitely often. In other words, we thus force every process to be selected to run infinitely often. Clearly this models a fair daemon.

4.2.3 Initial States

In order to determine whether or not the system is self-stabilizing, it is necessary to examine all possible initial states. SMV allows multiple initial states, and we can easily specify that the initial state can be any state.

For example, the state of a process is an integer ranging from 0 to $K-1$ in the *K*-state algorithm. We can specify that its initial value can take any value within the domain as follows ($K = 4$).

```
init(state) := {0, 1, 2, 3};
```

The above expression means that the possible initial values of `state` are 0, 1, 2, and 3. By specifying the initial values of all variables in this way, we can represent the fact that the system can take any initial state.

4.2.4 The Self-Stabilizing Property

As stated above, a self-stabilizing algorithm is defined as one that meets the convergence and closure properties. Now suppose that the predicate that identifies the legitimate states is expressed by CTL formula $legitimate$. Then,

- the convergence property holds iff CTL formula AF $legitimate$ holds in every global state, and
- the closure property holds iff CTL formula $legitimate \rightarrow$ AX $legitimate$ holds in every global state.

As a result, the self-stabilizing property is expressed by CTL formula AF $legitimate \wedge (legitimate \rightarrow$ AX $legitimate)$. (Note that the CTL formula is evaluated with respect to all initial states, i.e., all global states.)

Sometimes it is clear that the closure property holds from the definition of the legitimate states. In that case, we need to consider the convergence property only. In Section 5, we will discuss such cases.

In an SMV program, the property to be checked is preceded by the keyword `SPEC`, as follows. (`&` and `|` stand for logical **and** and logical **or**, respectively.)

```
SPEC AF legitimate & (legitimate
                        -> AX legitimate)
```

In the *K*-state algorithm, a global state is legitimate iff 1) there is exactly one privileged process in that state, and 2) every process will be eventually privileged in any computation starting with that state. Let $priv_i$ represent the fact that process p_i has a privilege. Each of the two conditions can be written in CTL as

$$\bigvee_{0 \le i \le n-1} (priv_i \wedge \bigwedge_{\substack{j \ne i \\ 0 \le j \le n-1}} \neg priv_j)$$

and

$$\bigwedge_{0 \le i \le n-1} \text{AF } priv_i,$$

respectively. In Fig. 2, the above two CTL formulae are denoted by symbols `condition1` and `condition2`, respectively. Hence, $legitimate$ can be written as `condition1 & condition2`.

4.3 Dealing with the Distributed Daemon

When the d-daemon is assumed, describing algorithms in the SMV language is slightly more complicated than the case of the c-daemon. Fig. 3 shows the SMV program for the *K*-state algorithm under the d-daemon.

To allow multiple processes to run at the same time, keyword `process` is not used in Fig. 3. In SMV, a module that is not associated with keyword `process` is always running. In other words, all processes are selected to run at any given time. Obviously, this is not adequate for representing the d-daemon.

To select an arbitrary set of processes to run, we use an additional variable `run` in each module. This variable takes a value of either `0` or `1`, and the value is randomly selected at any given time. Using the `case` expressions, we

```
MODULE main
VAR     p0 : type_p(p2.state);
        p1 : type_q(p0.state);
        p2 : type_q(p1.state);

DEFINE condition1 := (( p0.priv & !p1.priv & !p2.priv) |
                      (!p0.priv &  p1.priv & !p2.priv) |
                      (!p0.priv & !p1.priv &  p2.priv));
       condition2 := (AF p0.priv & AF p1.priv & AF p2.priv);
       legitimate :=  condition1 & condition2;

SPEC   AF legitimate & (legitimate -> AX legitimate)

MODULE type_p(L)
VAR     state : {0, 1, 2, 3};  run : boolean;
DEFINE  priv := (state = L);
ASSIGN  init(state) := {0, 1, 2, 3};  init(run) := {0, 1};
        next(state) :=  case
                          run & priv : (state + 1) mod 4;
                          1 : state;
                        esac;
        next(run) := {0, 1};
FAIRNESS  run

MODULE type_q(L)
VAR     state : {0, 1, 2, 3};  run : boolean;
DEFINE  priv := !(state = L);
ASSIGN  init(state) := {0, 1, 2, 3};  init(run) := {0, 1};
        next(state) := case
                         run & priv : L;
                         1 : state;
                       esac;
        next(run) := {0, 1};
FAIRNESS  run
```

Fig. 3. SMV program for the *K*-state algorithm under the d-daemon ($n = 3, K = 4$).

allow each process to actually run only when the value of `run` is 1.

We verified the *K*-state algorithm assuming $3 \leq n \leq 8$ and $K = n + 1$. Except for the case of $n = 8$ and the d-daemon, the verification was completed within a fairly admissible amount of time and the verification result showed that the self-stabilizing property holds. Table 1 shows the performance of the model checking procedure for this example in terms of the verification time, the maximum number of OBDD nodes used at any given time, and the number of nodes of the OBDD that represents the transition relation. (All measurements were performed on a Sun SS20 workstation with 160Mbyte memory. An NA in the table indicates that data was not collected since the verification was not completed within 10 hours.) The table also contains the number of the global states of the system (i.e., $|\mathcal{G}|$). For the *K*-state algorithm, it is given by K^n, since each process has K states.

In this table, one can see that when the number of processes is large, the size of the OBDD that represents the transition relation is extremely smaller than the size of global states. It can also be seen that the time and OBDD nodes used under the assumption of the d-daemon are much larger than the case of the c-daemon. This is because additional variables are used to model the d-daemon, thus leading to a larger state space to be explored.

5 Case Studies

5.1 Example 1: Mutual Exclusion in Special Networks

The proposed approach can also handle network topologies other than rings. Here we take Ghosh's mutual exclusion algorithm as an example [7]. This algorithm works in the special networks as shown in Fig. 4 ($m \geq 2$) and needs only two states, 0 and 1, per process. We let s_i denote the state of process i. The algorithm is presented below. The symbol b represents a binary value.

- process p_0
  if $((s_0, s_1) = (\neg b, b))\{s_0 := b;\}$
- process $p_{2i-1}(i = 1, 2, 3, \cdots, m-1)$
  if $((s_{2i-2}, s_{2i-1}, s_{2i}, s_{2i+1}) = (b, b, b, \neg b))\{s_{2i-1} := \neg b;\}$
- process $p_{2i}(i = 1, 2, 3, \cdots, m-1)$
  if $((s_{2i-2}, s_{2i-1}, s_{2i}, s_{2i+1}) = (b, b, \neg b, b))\{s_{2i} := b;\}$
- process p_{2m-1}
  if $((s_{2m-1}, s_{2m-2}) = (b, b))\{s_{2m-1} := \neg b;\}$.

Based on the verification approach presented in the previous section, we wrote SMV programs that represent the algorithm. Fig. 5 shows the SMV program for the c-daemon. The `main` module specifies the network topology with $m = 3$. (The `DEFINE` and `SPEC` parts in the `main` module are omitted, since they are the same as the *K*-state algorithm.)

We performed verification of this algorithm under both the c-daemon and the d-daemon. Table 2 shows the verification results and the performance of the model checking procedure for this example in terms of the verification time and the maximum number of OBDD nodes used at any given time. The table also contains the size of the OBDD that represents the transition relation and

TABLE 1
Verification Results and Performance for the *K*-State Algorithm ($K = n + 1$)

		$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
Result	c-daemon	true	true	true	true	true	true
	d-daemon	true	true	true	true	true	NA
Execution Time	c-daemon	0.1s	0.4s	4.6s	43.5s	285.2s	1836.0s
	d-daemon	0.1s	2.7s	36.9s	296.9s	2540.0s	NA
OBDD nodes used	c-daemon	631	5002	10432	18764	38697	185046
	d-daemon	2655	10458	27705	103378	511087	NA
Size of OBDD for transition relation	c-daemon	133	410	585	1026	1211	2605
	d-daemon	303	1410	3601	7122	11858	21923
# of global states		64	625	7776	117649	2097152	43046721

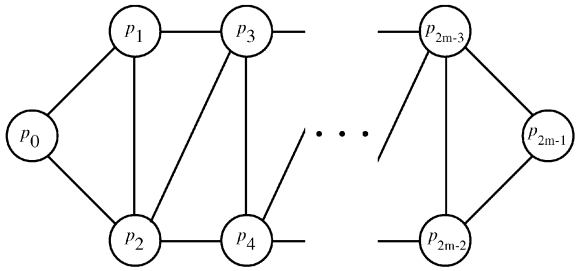


Fig. 4. Network topology where Ghosh's mutual exclusion algorithm works.

the number of global states of the system. For this algorithm, the number of the global states is given by 2^n since each process has only two states.

Unexpectedly, the time and the maximum size of OBDDs used under the d-daemon were smaller than those values used under the c-daemon, when the number of processes exceeded 10. The reason is that in the case of the d-daemon, the transition relation has a relatively compact OBDD representation.

This phenomenon is consistent with the results of comparing two plausible models of asynchronous circuits [16]. These two models are called the *interleaving model* and the *simultaneous model*. In the former model, only one state component changes value in a given transition, while any or all state variables may change state in the latter model. They are thus analogous to the c-daemon and the d-daemon, respectively. In [16], it is shown that OBDD-based techniques tend to perform better on the simultaneous model, especially when the number of variables is large. Since the addition of variable `run`, which is unnecessary for the c-daemon model, degrades the verification performance, this phenomenon was observed only when the number of processes was sufficiently large.

Unlike Ghosh's algorithm, other self-stabilizing algorithms discussed elsewhere in the paper do not have such similarities to hardware circuits, since variables in these algorithms have a much larger domain than a Boolean variable has. Actually, such a phenomenon was not observed in the verification of these algorithms.

5.2 Example 2: Leader Election on Uniform Rings

Both of the two algorithms discussed before are used for achieving mutual exclusion. In the rest of the section, we show that the proposed approach can also be applied to algorithms used to solve other problems.

```
MODULE main
VAR  p0 : process type_p(p1.state);
     p1 : process type_q(p0.state,p2.state,p3.state);
     p2 : process type_r(p0.state,p1.state,p3.state);
     p3 : process type_q(p2.state,p4.state,p5.state);
     p4 : process type_r(p2.state,p3.state,p5.state);
     p5 : process type_s(p4.state);

MODULE type_p(a)
DEFINE  priv := (state = !a);
VAR     state : {0, 1};
ASSIGN  init(state) := {0, 1};
        next(state) := case  priv: a;  1 : state; esac;
FAIRNESS  running

MODULE type_q(a,c,d)
VAR  state : {0, 1};
DEFINE  priv := ((state = a) & (state = c) & !(state = d));
ASSIGN  init(state) := {0, 1};
        next(state) := case priv : !state ; 1 : state ; esac;
FAIRNESS  running

MODULE type_r(a,b,d)
VAR  state : {0, 1};
DEFINE priv := (!(state = a) &  !(state = b) & !(state = d));
ASSIGN  init(state) := {0, 1};
        next(state) := case priv: !state ; 1 : state ;  esac;
FAIRNESS  running

MODULE type_s(a)
VAR  state : {0, 1};
DEFINE priv := (state = a);
ASSIGN  init(state) := {0, 1};
        next(state) := case  priv: !state; 1 : state; esac;
FAIRNESS  running
```

Fig. 5. An SMV program for Ghosh's mutual exclusion algorithm ($m = 3, n = 6$).

In this section, we discuss the *leader election problem* on rings, which is the problem of selecting one process as a leader on a ring where no distinguished process initially exists. Consider a ring that consists of n processes, $p_0, p_1, \cdots, p_{n-1}$, that are connected in this order; and assume that the ring is *uniform*; that is, all the processes on the ring have no identifiers and execute the same algorithm. Subscripts are thus used only for explanation purposes, and processes cannot make use of them.

In [13], Huang proposed a self-stabilizing leader election algorithm that works on rings of primal size under the c-daemon. In the algorithm, the state of each process is in $\{0, 1, 2, \cdots, n-1\}$. A process is considered to be a leader iff the state is 0. For any process p_i, we use the symbols S, L,

TABLE 2
Verification Results and Performance for Ghosh's Mutual Exclusion Algorithm

		$n = 4$	$n = 6$	$n = 8$	$n = 10$	$n = 12$	$n = 14$
Result	c-daemon	true	true	true	true	true	true
	d-daemon	true	true	true	true	true	true
Execution Time	c-daemon	0.1s	0.4s	3.1s	22.9s	182.0s	1161.5s
	d-daemon	0.1s	0.6s	3.0s	12.7s	55.9s	202.9s
OBDD nodes used	c-daemon	955	7111	10017	10483	11610	20208
	d-daemon	2120	10008	10161	10413	10962	12023
Size of OBDD for transition relation	c-daemon	58	111	176	253	342	443
	d-daemon	55	88	121	154	187	220
# of global states		16	64	256	1024	2048	4096

```
MODULE main
VAR     p0 : process type_p(p2.state,p1.state);
        p1 : process type_p(p0.state,p2.state);
        p2 : process type_p(p1.state,p0.state);

DEFINE  legit0 := AG( p0.leader & !p1.leader & !p2.leader);
        legit1 := AG(!p0.leader &  p1.leader & !p2.leader);
        legit2 := AG(!p0.leader & !p1.leader &  p2.leader);

SPEC    AF(legit0 | legit1 | legit2)

MODULE type_p(L,R)
VAR     state : {0, 1, 2};
DEFINE  leader := (state = 0);
        x := case
               (L = state) : 3;
               1 : (state - L) mod 3;
             esac;
        y := case
               (state = R) : 3;
               1 : (R - state) mod 3;
             esac;
ASSIGN  init(state) := {0, 1, 2};
        next(state) := case
                         (x = y) & (y = 3) : (state + 1) mod 3;
                         (x < y) : (state + 1) mod 3;
                         1 : state;
                       esac;
FAIRNESS  running
```

Fig. 6. An SMV program for the leader election algorithm ($n = 3$).

and R to denote its state, the state of its left neighbor p_{i-1}, and the state of its right neighbor p_{i+1}, respectively. Let $x = g(L, S)$ and $y = g(S, R)$, where

$$g(a,b) = \begin{cases} n & a = b \\ (b-a) \bmod n & \text{otherwise.} \end{cases}$$

Then the leader election algorithm is as follows:

- process $p_i (i = 0, 1, 2, \cdots, n-1)$
  if $(x = y$ and $y = n)$ $\{S := S + 1 \bmod n;\}$
  if $(x < y)$ $\{S := S + 1 \bmod n;\}$.

We define a global state to be legitimate iff 1) in that state, there is exactly one process (say p_i) such that p_i is a leader and other $n-1$ processes are not a leader, and 2) this property will always hold at any state in every computation starting with the state. A leader election algorithm is considered self-stabilizing iff the system that runs the algorithm reaches a legitimate state regardless of the initial state. Note that the closure property is already taken into consideration in the definition of the legitimate states.

Now let $leader_i$ be the predicate that is true iff only p_i is a leader. By definition, the legitimate states are the states in which AG $leader_0 \vee$ AG $leader_1 \vee \cdots \vee$ AG $leader_{n-1}$ holds. The self-stabilizing property can then be written in CTL as AF (AG $leader_0 \vee$ AG $leader_1 \vee \cdots \vee$ AG $leader_{n-1}$). Fig. 6 shows an SMV program that describes this algorithm when $n = 3$.

Although the algorithm assumes the c-daemon and rings of primal size, we verified the algorithm in the case of the d-daemon and/or rings of composite size, in order to demonstrate how SMV works when a given correctness property does not hold. Table 3 shows the results of the verification. (Note that Huang proved that no uniform, deterministic self-stabilizing leader algorithm exists if n is composite [13].) These results show that the algorithm does not work under the d-daemon even if n is prime. When an SMV program does not meet a given property to be checked, SMV provides a computation path on which the property does not hold. In the case of $n = 3$ and the d-daemon, for example, SMV detected the following computation, which never reaches a legitimate state.

$$(2,2,2) \overset{\{p_1,p_2,p_3\}}{\rightarrow} (0,0,0) \overset{\{p_1,p_2,p_3\}}{\rightarrow} (2,2,2) \overset{\{p_1,p_2,p_3\}}{\rightarrow} (0,0,0) \overset{\{p_1,p_2,p_3\}}{\rightarrow} \cdots$$

3Table 3 also shows the performance of the model checking procedure and the number of the global states, which is n^n since each process has n local states in this algorithm.

5.3 Example 3: Ring Orientation

The next problem we consider is *ring orientation*, which is the problem of orienting a ring in one direction where each node has no sense of direction. We assume that each process p_i cannot tell which of its two adjacent processes is p_{i-1} or p_{i+1} and that the ring is uniform as in the previous example.

During the execution of a ring orientation algorithm, each process chooses one of its adjacent processes as the forward process and the other as the backward process. We denote the forward process of p_i by $Forw(p_i)$.

Let $AP1$ and $AP2$ denote the two processes adjacent to p_i. We assume that each process p_i has a variable $dir \in \{B, F\}$

TABLE 3
Verification Results and Performance for the Leader Election Algorithm

		$n=3$	$n=4$	$n=5$	$n=6$	$n=7$	$n=8$
Result	c-daemon	**true**	false	**true**	false	**true**	NA
	d-daemon	false	false	false	false	NA	NA
Execution Time	c-daemon	0.0s	0.7s	23.0s	38.5s	40363.1s	NA
	d-daemon	0.1s	1.9s	83.9s	166.2s	NA	NA
OBDD nodes used	c-daemon	860	10008	19628	20290	1438017	NA
	d-daemon	2182	10087	53137	124699	NA	NA
Size of OBDD for transition relation	c-daemon	125	326	1261	2216	4273	5932
	d-daemon	127	1058	6814	38322	129090	350675
# of global states		27	256	3125	46656	823543	16777216

```
MODULE main
VAR  p0 : process p(p2.s, p2.t, p1.s, p1.t);
     p1 : process p(p0.s, p0.t, p2.s, p2.t);
     p2 : process p(p1.s, p1.t, p0.s, p0.t);

DEFINE legitimate :=  AG(p0.des & p1.des & p2.des) | AG(!p0.des & !p1.des & !p2.des);

SPEC  AF legitimate

MODULE p(in1_s,in1_t,in2_s,in2_t)
VAR   s:{0,1}; t:{0,1}; dir:{B,F}; AP1_pc:{-1,1};

DEFINE  des := ((dir = B)&(AP1_pc = -1)) | ((dir = F)&(AP1_pc = 1));
  S1 := case  (AP1_pc = -1) : in1_s; 1 : in2_s;  esac;
  T1 := case  (AP1_pc = -1) : in1_t; 1 : in2_t;  esac;
  S2 := case  (AP1_pc = -1) : in2_s; 1 : in1_s;  esac;
  T2 := case  (AP1_pc = -1) : in2_t; 1 : in1_t;  esac;

  r1 := (S1 = S2);
  r2 := (S1 = s) & (S2 = !s) & (T2 = t) & (T1 = !t) & (t = 1);
  r3 := (S2 = s) & (S1 = !s) & (T1 = t) & (T2 = !t) & (t = 1);
  r4 := ((S1 = s) & (S2 = !s) & (T1 = t)) | ((S2 = s) & (S1 = !s) & (T2 = t));

ASSIGN  init(s):={0,1}; init(t):={0,1}; init(dir):={B,F}; init(AP1_pc):={-1,1};

  next(s) :=   case  r1 : !S1;  r2 | r3      : !s;  1: s;    esac;
  next(t) :=   case  r1 : 1;    r2 | r3 | r4 : !t;  1: t;    esac;
  next(dir) := case  r2 : F;    r3 : B;             1: dir;  esac;
  next(AP1_pc) := AP1_pc;

FAIRNESS  running
```

Fig. 7. An SMV program for the ring orientation algorithm [11] ($n = 3$).

to represent its decision in such a way that $Forw(p_i) = AP1$ iff $dir = B$ and $Forw(p_i) = AP2$ iff $dir = F$. Then we say that a ring is oriented iff exactly one of the following two conditions holds: (Condition 1) $Forw(p_i) = p_{i-1}$ for all $i = 0, 1, \cdots, n-1$, or (Condition 2) $Forw(p_i) = p_{i+1}$ for all $i = 0, 1, \cdots, n-1$.

In [11], Hoepman proposed uniform self-stabilizing ring-orientation algorithms for rings of odd size both for the state-reading model and the link-register model. In this paper, we take the algorithm for the state-reading model. In the algorithm, each process has two Boolean variables, S and T, in addition to dir. The following is such an algorithm where $S1$ and $T1$ denote the values of S and T of $AP1$, and similarly, $S2$ and $T2$ denote the values of S and T of $AP2$.

- process $p_i (i = 0, 1, 2, \cdots, n-1)$
  if $(S1 = S2)\{S := \neg S1; T := 1;\}$
  if $(S1 = S = \neg S2$ and $\neg T1 = T = T2 = 1)$ $\{S := \neg S;$ $T := 0;$ $dir := F;\}$
  if $(\neg S1 = S = S2$ and $T1 = T = \neg T2 = 1)$ $\{S := \neg S;$ $T := 0;$ $dir := B;\}$
  if $((S1 = S = \neg S2$ and $T1 = T)$ or $(\neg S1 = S = S2$ and $T = T2))$ $\{T := \neg T;\}$.

In the ring orientation problem, a global state is legitimate iff 1) in that state the ring is oriented in one direction, i.e., one of the above two conditions holds, and 2) the ring will be oriented in the same direction at any state in every computation starting with that state. A ring orientation algorithm is self-stabilizing iff it reaches a legitimate state from any initial state.

Let $Condition1$ ($Condition2$) be true iff Condition 1 (Condition 2) holds. The legitimate states can then be defined as those where AG $Condition1 \vee$ AG $Condition2$ holds. Hence, the self-stabilizing property is written in CTL as AF (AG $Condition1 \vee$ AG $Condition2$).

Fig. 7 shows an SMV program that represents the ring orientation algorithm when $n = 3$. As stated above, each

TABLE 4
Verification Results and Performance for the Ring-Orientation Algorithm [11]

		$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$
Result	c-daemon	**true**	false	**true**	false	NA
	d-daemon	false	false	false	false	NA
Execution Time	c-daemon	1.3s	4.6s	128.1s	1010.9s	NA
	d-daemon	3.8s	13.0s	534.0s	4702.8s	NA
OBDD nodes used	c-daemon	10233	12212	62364	105115	NA
	d-daemon	10677	23330	125351	1160540	NA
Size of OBDD for transition relation	c-daemon	448	758	1080	1414	1760
	d-daemon	756	2789	6980	13978	21924
# of global states		512	4096	32768	262144	2097152

```
1   /* Read input registers */
2   read(RI1, (l1, d1)); read(RI2, (l2, d2));
3
4   if (d1 = F) {
5     if (dir = F and label = l1 and label ≠ H)
6       label := 1 − l1; /* Rule 1a */
7     else if (dir = B) {
8       if ((label ≠ l1 and l1 = H) or (label = 0 and l1 = 1))
9         {label := H; dir := F; } /* Rule 2a */
10      else if (label = l1 and label = H)
11        label := 0; /* Rule 3a */
12    }
13  }
14  else if (d1 = B and ((dir = B and label = H) /* Rule 4a */
15        or (dir = F and label ≠ 0))) /* Rule 5a */
16    label := 0;
17
18  if (d2 = F) {
19    if (dir = B and label = l2 and label ≠ H)
20      label := 1 − l2; /* Rule 1b */
21    else if (dir = F) {
22      if ((label ≠ l2 and l2 = H) or (label = 0 and l2 = 1))
23        {label := H; dir := B; } /* Rule 2b */
24      else if (label = l2 and label = H)
25        label := 0; /* Rule 3b */
26    }
27  }
28  else if (d2 = B and ((dir = F and label = H) /* Rule 4b */
29        or (dir = B and label ≠ 0))) /* Rule 5b */
30    label := 0;
31
32  /* Write output registers */
33  if (dir = F) {write(RO1, (label, B)); write(RO2, (label, F)); }
34  else {write(RO1, (label, F)); write(RO2, (label, B)); }
```

Fig. 8. The ring orientation algorithm [26].

process p_i does not know which of its two adjacent processes, $AP1$ and $AP2$, is p_{i-1} and which is p_{i+1}. This fact means that it is necessary to consider two cases for each process p_i in verification, that is, the case where $AP1 = p_{i-1}$ and $AP2 = p_{i+1}$ and the case where $AP1 = p_{i+1}$ and $AP2 = p_{i-1}$. Thus, we have to check a total of 2^n cases to verify the algorithm.

In order to handle the 2^n cases at a time, we add a special variable AP1_pc($AP1\_pc$) to each process in the SMV program shown in Fig. 7. In the SMV program, the variable takes a value of either -1 or 1. If $AP1\_pc$ for process p_i has a value of -1, then $AP1 = p_{i-1}$ and $AP2 = p_{i+1}$; otherwise, $AP1 = p_{i+1}$ and $AP2 = p_{i-1}$. By allowing nondeterministic choice of the initial value of $AP1\_pc$, we can verify all the 2^n cases at one try. Note that $Forw(p_i) = p_{i-1}$ holds iff $dir = B \wedge AP1\_pc = -1$ or $dir = F \wedge AP1\_pc = 1$. Let predicate des_i be true iff $Forw(p_i) = p_{i-1}$ holds. Then Condition 1 can be written as $des_0 \wedge des_1 \wedge \cdots \wedge des_{n-1}$, while Condition 2 is given by $\neg des_0 \wedge \neg des_1 \wedge \cdots \wedge \neg des_{n-1}$.

Although the algorithm assumes the c-daemon and rings with an odd number of processes, we verified the algorithm in the case of the d-daemon and/or rings with an even number of processes. Table 4 shows the results of verification. Note that no uniform and deterministic self-stabilizing ring orientation algorithm exists if n is even [14]. In [14], it is also proven that no uniform and deterministic self-stabilizing ring orientation algorithm exists under the d-daemon in the state-reading model. As shown in the table, the verification results are consistent with this impossibility result.

This table also shows the performance of the model checking procedure for this example in terms of the verification time and the maximum number of OBDD nodes used at any given time. The table also contains the size of the OBDD for the transition relation and the number of the global states of the system, which is 8^n since each process has eight local states.

5.4 Example 4: Ring Orientation in the Link-Register Model

The four algorithms discussed above assume the state-reading model, in which processes can read the states of other processes directly. Here we take an algorithm that works in the link-register model. The algorithm, which is proposed by Umemoto et al. in [26], is also for ring orientation. This algorithm is designed to run on rings of odd size under the d-daemon. (Note that no deterministic ring orientation algorithm exists when n is even.) As described before, communication is done by means of registers in the link-register model. Since we assume that the topology of the system is a ring, there are a total of $2n$ registers (i.e., $R_{0,1}, R_{1,0}, R_{1,2}, R_{2,1}, \cdots, R_{n-1,0}, R_{0,n-1}$).

Each process has two adjacent processes $AP1$ and $AP2$. For each adjacent process $AP1$ ($AP2$), we denote its output register by $RO1$ ($RO2$) and its input register by $RI1$ ($RI2$).

Fig. 8 shows the algorithm. As shown in the figure, this algorithm works according to two sets of five rules. The state of each component (process and register) is a tuple $(label, dir)$, where $label \in \{0, 1, H\}$ and $dir \in \{F, B\}$. Thus each component has six states. Processes selected to run read $RI1$ and $RI2$, change their own state, and update $RO1$ and $RO2$ atomically. In Fig. 8, instruction "read(R, v)" reads input register R and stores its contents in v, and "write(R, v)" writes the contents of local variable v to output register R.

Fig. 9 shows an SMV program that describes the ring-orientation algorithm under the d-daemon. In the SMV program, module p specifies the behavior of each process p_i and its output registers $R_{i,i-1}$ and $R_{i,i+1}$. We use the same technique as the previous example in order to model the fact that for each process p_i two distinct situations can occur, that is, $AP1$ can be either p_{i-1} or p_{i+1}.

Table 5 shows the results of verifying this algorithm. The results indicate that the algorithm does not work under the d-daemon. By examining a counterexample that SMV produced, we found that there is a fair computation that never reaches the legitimate states. Fig. 10 shows part of one such a computation. Here for every process p_i, $AP1 = p_{i+1}$ and $AP2 = p_{i-1}$ are assumed. In this part of the computation, g_1 and g_{13} are the same global state, and there is no

```
MODULE main
VAR  p0 : p(p2.out2_label,p2.out2_dir,p1.out1_label,p1.out1_dir);
     p1 : p(p0.out2_label,p0.out2_dir,p2.out1_label,p2.out1_dir);
     p2 : p(p1.out2_label,p1.out2_dir,p0.out1_label,p0.out1_dir);

DEFINE legitimate :=  AG(p0.des & p1.des & p2.des) | AG(!p0.des & !p1.des & !p2.des);

SPEC  AF legitimate

MODULE p(in1_label,in1_dir,in2_label,in2_dir)

VAR label : {0,1,H};  dir : {F,B}; AP1_pc : {-1,1}; run : boolean;
    ro1_label : {0,1,H};  ro1_dir : {F,B}; -- output register 1
    ro2_label : {0,1,H};  ro2_dir : {F,B}; -- output register 2

DEFINE
  des := ((dir = B)&(AP1_pc = -1)) | ((dir = F)&(AP1_pc = 1));

  l1 := case (AP1_pc = -1) : in1_label; 1 : in2_label; esac;
  d1 := case (AP1_pc = -1) : in1_dir;   1 : in2_dir;   esac;
  l2 := case (AP1_pc = -1) : in2_label; 1 : in1_label; esac;
  d2 := case (AP1_pc = -1) : in2_dir;   1 : in1_dir;   esac;

  out1_label := case (AP1_pc = -1) : ro1_label; 1 : ro2_label; esac;
  out1_dir   := case (AP1_pc = -1) : ro1_dir;   1 : ro2_dir;   esac;
  out2_label := case (AP1_pc = -1) : ro2_label; 1 : ro1_label; esac;
  out2_dir   := case (AP1_pc = -1) : ro2_dir;   1 : ro1_dir;   esac;

  tmp_label := case
    d1 = F & dir = F & label = l1 & label = 0 : 1;  -- Rule 1a
    d1 = F & dir = F & label = l1 & label = 1 : 0;  -- Rule 1a
    (d1 = F & dir = B) & ((!(label = l1) & l1 = H) | (label = 0 & l1 = 1)) : H;   -- Rule 2a
    d1 = F & dir = B & label = l1 & label = H : 0;  -- Rule 3a
    d1 = B & ((dir = B & label = H) | (dir = F & !(label = 0))) : 0;   -- Rules 4a,5a
    1 : label; esac;

  tmp_dir := case
    (d1 = F & dir = B) & ((!(label = l1) & l1 = H) | (label = 0 & l1 = 1)) : F;   -- Rule 2a
    1 : dir; esac;

ASSIGN init(label):={0,1,H};  init(dir):={F,B}; init(AP1_pc):={-1,1}; init(run):={0,1};
  init(ro1_label):={0,1,H}; init(ro1_dir):={F,B};
  init(ro2_label):={0,1,H}; init(ro2_dir):= {F,B};

  next(label) := case !run : label;
    d2 = F & tmp_dir = B & tmp_label = l2 & tmp_label = 0 : 1;  -- Rule 1b
    d2 = F & tmp_dir = B & tmp_label = l2 & tmp_label = 1 : 0;  -- Rule 1b
    (d2 = F & tmp_dir = F) & ((!(tmp_label = l2) & l2 = H) | (tmp_label = 0 & l2 = 1)) : H;  -- Rule 2b
    d2 = F & tmp_dir = F & tmp_label = l2 & tmp_label = H : 0;  -- Rule 3b
    d2 = B & ((tmp_dir = F & tmp_label = H) | (tmp_dir = B & !(tmp_label = 0))) : 0;  -- Rules 4b,5b
    1 : tmp_label; esac;

  next(dir):= case !run : dir;
    (d2 = F & tmp_dir = F) & ((!(tmp_label = l2) & l2 = H) | (tmp_label = 0 & l2 = 1)) : B;  -- Rule 2b
    1 : tmp_dir; esac;

  next(ro1_label):= case run : next(label); 1: ro1_label; esac;
  next(ro1_dir)  := case !run : ro1_dir; next(dir) = F : B; 1 : F; esac;
  next(ro2_label):= case run : next(label); 1: ro2_label; esac;
  next(ro2_dir)  := case !run : ro2_dir; next(dir) = F : F; 1 : B; esac;

  next(AP1_pc):= AP1_pc; next(run) := {0,1};

FAIRNESS  run
```

Fig. 9. An SMV program for the ring orientation algorithm [26] ($n = 3$).

process that is not selected to run. Thus, one can see that $g_1\ g_2\ \cdots\ g_{12}\ g_1\ g_2\ \cdots\ g_{12}\ g_1\ g_2\ \cdots$ is a fair computation and does not reach any legitimate state.

We investigated the cause of such livelock and found that it may occur when Rule 2a and Rule 2b, which are the only rules that can change the value of dir, are applied to two neighboring processes at the same time. In [26], livelock freedom is proved based on the fact that such a case never occurs (Lemma 4). Therefore, the proof does not hold for the original algorithm. For example, consider the transition $g_2 \rightarrow g_3$ in Fig. 10. In this transition, p_0 changes its direction according to Rule 2a, while for p_1, two rules are applied consecutively. First, Rule 5a is applied, which means that $label = 0$ and $dir = F$ hold temporarily. Then Rule 2b is applied and the state of p_2 finally becomes (H, B).

One way to prevent the occurrence of such a situation is to ensure that no more than one rule is applied to each process at a time, and this makes Lemma 4 in [26] hold. Fig. 11 shows the corrected algorithm. (Only lines different from Fig. 8 are shown.) For example, only Rule 4a can be

TABLE 5
Verification Results and Performance for the Ring-Orientation Algorithm [26]

		Original Algorithm		Corrected Algorithm	
		$n=3$	$n=4$	$n=3$	$n=4$
Result	c-daemon	true	NA	true	NA
	d-daemon	**false**	NA	**true**	NA
Execution Time	c-daemon	411.6s	NA	238.6s	NA
	d-daemon	15642.0s	NA	2737.1s	NA
OBDD nodes used	c-daemon	529558	NA	363986	NA
	d-daemon	2119169	NA	1287705	NA
Size of OBDD for transition relation	c-daemon	3123	4439	2711	3999
	d-daemon	212069	765141	153837	642169
# of global states		10077696	2176782336	10077696	2176782336

	p_0	p_1	p_2	$R_{0,1}$	$R_{1,0}$	$R_{1,2}$	$R_{2,1}$	$R_{2,0}$	$R_{0,2}$	processes to make move
g_1	(H,B)	(H,F)	$(0,B)$	(H,F)	(H,F)	(H,B)	$(0,B)$	$(0,F)$	(H,B)	p_0
g_2	$(1,B)$	(H,F)	$(0,B)$	$(1,F)$	(H,F)	(H,B)	$(0,B)$	$(0,F)$	$(1,B)$	p_0, p_1
g_3	(H,F)	(H,B)	$(0,B)$	(H,B)	(H,B)	(H,F)	$(0,B)$	$(0,F)$	(H,F)	p_2
g_4	(H,F)	(H,B)	$(0,F)$	(H,B)	(H,B)	(H,F)	$(0,F)$	$(0,B)$	(H,F)	p_1, p_2
g_5	(H,F)	$(0,B)$	(H,B)	(H,B)	$(0,B)$	$(0,F)$	(H,B)	(H,F)	(H,F)	p_2
g_6	(H,F)	$(0,B)$	$(1,B)$	(H,B)	$(0,B)$	$(0,F)$	$(1,B)$	$(1,F)$	(H,F)	p_0, p_2
g_7	(H,B)	$(0,B)$	(H,F)	(H,F)	$(0,B)$	$(0,F)$	(H,F)	(H,B)	(H,B)	p_1
g_8	(H,B)	$(0,F)$	(H,F)	(H,F)	$(0,F)$	$(0,B)$	(H,F)	(H,B)	(H,B)	p_0, p_1
g_9	$(0,B)$	(H,B)	(H,F)	$(0,F)$	(H,B)	(H,F)	(H,F)	(H,B)	$(0,B)$	p_1
g_{10}	$(0,B)$	$(1,B)$	(H,F)	$(0,F)$	$(1,B)$	$(1,F)$	(H,F)	(H,B)	$(0,B)$	p_1, p_2
g_{11}	$(0,B)$	(H,F)	(H,B)	$(0,F)$	(H,F)	(H,B)	(H,B)	(H,F)	$(0,B)$	p_0
g_{12}	$(0,F)$	(H,F)	(H,B)	$(0,B)$	(H,F)	(H,B)	(H,B)	(H,F)	$(0,F)$	p_0, p_2
g_{13}	(H,B)	(H,F)	$(0,B)$	(H,F)	(H,F)	(H,B)	$(0,B)$	$(0,F)$	(H,B)	—

Fig. 10. Livelock.

applied to p_1 at g_2 in the corrected algorithm. Using the proposed approach, we verified that this algorithm works correctly under the d-daemon when $n=3$. Fig. 12 shows the SMV program that represents the algorithm. (In this figure, only `next(label)` and `next(dir)` are shown since the remaining part is the same as in Fig. 9, except that `tmp_label` and `tmp_dir` can be omitted.)

6 Verifying Self-Stabilization Using SPIN

For comparison purposes, we present the results of using another model checker, called SPIN [12], to verify a self-stabilizing algorithm in this section. SPIN is a very fast model checker based on explicit state enumeration, and like SMV, it is widely available.

```
4   if (d_1 = F) {
5     if (dir = F and label = l_1 and label ≠ H)
6       {label := 1 − l_1; goto JUMP;}/* Rule 1a */
7     else if (dir = B) {
8       if ((label ≠ l_1 and l_1 = H) or (label = 0 and l_1 = 1))
9         {label := H; dir := F; goto JUMP;} /* Rule 2a */
10      else if (label = l_1 and label = H)
11        {label := 0; goto JUMP;}/* Rule 3a */
12    }
13  }
14  else if (d_1 = B and ((dir = B and label = H) /* Rule 4a */
15      or (dir = F and label ≠ 0))) /* Rule 5a */
16    {label := 0; goto JUMP;}

31  JUMP:
```

Fig. 11. The corrected algorithm.

In addition to various techniques for efficient verification, SPIN incorporates a different state reduction approach than symbolic representation. This approach, called *partial order reduction*, has been proven to be very successful in verifying concurrent systems and communication protocols [8], [12], [27]. It is based on the observation that the validity of a given correctness property is often insensitive to the order in which current and independently executed events are interleaved. Given an initial state, these techniques generate a reduced set of reachable states that is indistinguishable for the given property, instead of generating the whole reachable state space.

The input language for SPIN is called PROMELA; Fig. 13 shows a PROMELA program for the Dijkstra's *K*-state mutual exclusion algorithm under the c-daemon. This program is a modification of the one proposed in [23]. Since the original program only modeled the algorithm under the condition that an initial state is given, we added if statements to enforce each process to nondeterministically change its state in the first move of the process. Also, an initial state must be specified in a PROMELA program, therefore this program sets the initial state of each process to 0. Due to these modifications, however, the algorithm is allowed to start with any global state. Unfortunately, the modifications do not preserve the closure property. In the

```
next(label) := case
  !run : label;
  d1 = F & dir = F & label = l1 & label = 0 : 1;  -- Rule 1a
  d1 = F & dir = F & label = l1 & label = 1 : 0;  -- Rule 1a
  (d1 = F & dir = B) & ((!(label = l1) & l1 = H) | (label = 0 & l1 = 1)) : H;  -- Rule 2a
  d1 = F & dir = B & label = l1 & label = H : 0;  -- Rule 3a
  d1 = B & ((dir = B & label = H) | (dir = F & !(label = 0))) : 0;  -- Rules 4a,5a
  d2 = F & dir = B & label = l2 & label = 0 : 1;  -- Rule 1b
  d2 = F & dir = B & label = l2 & label = 1 : 0;  -- Rule 1b
  (d2 = F & dir = F) & ((!(label = l2) & l2 = H) | (label = 0 & l2 = 1)) : H;  -- Rule 2b
  d2 = F & dir = F & label = l2 & label = H : 0;  -- Rule 3b
  d2 = B & ((dir = F & label = H) | (dir = B & !(label = 0))) : 0;  -- Rules 4b,5b
  1 : label;
esac;

next(dir):= case
  !run : dir;
  (d1 = F & dir = B) & ((!(label = l1) & l1 = H) | (label = 0 & l1 = 1)) : F;  -- Rule 2a
  (d2 = F & dir = F) & ((!(label = l2) & l2 = H) | (label = 0 & l2 = 1)) : B;  -- Rule 2b
  1: dir;
esac;
```

Fig. 12. An SMV program for the corrected algorithm.

rest of this section, therefore, we limit our discussion to verification of the convergence property.

SPIN adopts *Linear Time Logic* (LTL) [18] to specify the property to be verified. Since the CTL formula used for representing the legitimate states of the *K*-state algorithm is not expressible in LTL, to signify the convergence property we use LTL formula AF $legitimate$, where

$$legitimate = \bigvee_{0 \le i \le n-1} (priv_i \wedge \bigwedge_{\substack{j \ne i \\ 0 \le j \le n-1}} \neg priv_j) \wedge \bigwedge_{0 \le i \le n-1} \mathrm{F}\ priv_i.$$

```
#define n 3
#define K 4

byte state[n];

proctype P(int index){
if
:: state[index] = 0
:: state[index] = 1
:: state[index] = 2
:: state[index] = 3
fi;
do
:: atomic{state[index] == state[(n+index-1)% n] ->
            state[index] = (state[index]+1) % K }
od
}

proctype Q(int index){
if
:: state[index] = 0
:: state[index] = 1
:: state[index] = 2
:: state[index] = 3
fi;
do
:: atomic{state[index] != state[(n+index-1)% n] ->
            state[index] = state[(n+index-1) % n] }
od
}

init{
    atomic{state[0]=0; state[1]=0; state[2]=0};
    atomic{run P(0); run Q(1); run Q(2)}
}
```

Fig. 13. A PROMELA program for the *K*-state algorithm under the c-daemon ($n = 3, K = 4$).

(For the formal definition of LTL, the readers are referred to, for example, [4].)

We applied SPIN to the *K*-state algorithm with and without enabling partial order reduction. Table 6 shows the verification times of SPIN. An NA in the table indicates that the verification was not completed due to memory shortage. The results show that the use of SPIN is not feasible unless the number of processes is small, and that it is more vulnerable to the state explosion problem than symbolic model checking. It can also be seen that for this example partial order reduction did not work effectively and even worsened the performance. By comparing the results with those presented in Table 1, we conclude that the proposed method is superior in terms of verification performance.

Nevertheless, we expect that SPIN can be more useful than SMV for verification of self-stabilizing communication protocols (e.g., [10], [24]), because typically in such protocols, only two processes are involved, and communications between processes are implemented by message passing. In SPIN, such communication can easily be modeled by using communication commands in PROMELA. Although this topic is beyond the scope of the paper, we consider it one of the possible directions of future study.

7 Conclusions

In this paper, we proposed to use symbolic model checking to verify distributed algorithms against the self-stabilizing property. We presented an approach in which the SMV system can be used for this purpose, and showed the effectiveness of the proposed approach, by using it to verify several algorithms. During the verification process, we found an error in one of these algorithms. Due to the nature of model checking, the proposed approach is applicable only when the number of processes is modest. However, we believe that this approach is useful for designing self-stabilizing algorithms, since, as demonstrated in this paper,

TABLE 6
Verification Times of SPIN for the *K*-State Algorithm ($K = n + 1$)

	$n = 3$	$n = 4$	$n = 5$	$n = 6$	$n = 7$	$n = 8$
without partial order reduction	0.1s	0.5s	11.9s	603.8s*	NA	NA
with partial order reduction	0.1s	0.5s	12.6s	608.6s*	NA	NA
# of global states	64	625	7776	117649	2097152	43046721

(*: Compression was used to avoid memory shortage.)

it can help designers to detect and correct errors in the algorithms.

There are many directions for future work. For example, applicability of model checking techniques other than symbolic model checking need to be further examined; in particular, the use of symmetry seems likely to be effective for state reduction, because self-stabilizing systems appearing in the literature frequently exhibit considerable symmetry (uniform systems [11], [13], [14], [26] are such typical examples).

Although model checking has an advantage because it can be performed automatically, there will always be situations where theorem proving is required for complete verification, since model checking can only be applied to finite state systems. A new research direction in formal verification attempts to combine model checking and mechanical theorem proving (e.g., [21]). Application of this new approach to self-stabilizing systems also deserves further study.

Recently, some unique techniques have been proposed to reason about self-stabilizing systems; for example, in [25], control theory is applied for this purpose. In [1], the use of string rewriting systems is suggested to model and verify self-stabilizing rings. Extension of these approaches, including their automation, is also an interesting research topic.

ACKNOWLEDGMENTS

The authors would like to thank Ms. K. Teresa Khidir for her helpful comments and Dr. Masahide Nakamura for his support. The authors also thank the anonymous referees for their helpful suggestions on how to improve this paper.

REFERENCES

[1] J. Beauquier, B. Bérard, and L. Fribourg, "A New Rewrite Method for Proving Convergence of Self-Stabilizing Systems," *Proc. 13th Int'l Symp. Distributed Computing (DISC '99)*, pp. 240-253, 1999.
[2] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond," *Information and Computation*, vol. 98, no. 2, pp. 142-170, 1992.
[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal-Logic Specifications," *ACM Trans. Programming Languages and Systems*, vol. 8, no. 2, pp. 244-263, 1986.
[4] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*. MIT Press, 1999.
[5] E.W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Comm. ACM*, vol. 17, no. 11, pp. 643-644, Nov. 1974.
[6] E.W. Dijkstra, "A Belated Proof of Self-Stabilization," *Distributed Computing*, vol. 1, no. 1, pp. 5-6, 1986.
[7] S. Ghosh, "Binary Self-Stabilization in Distributed Systems," *Information Processing Letters*, vol. 40, no. 3, pp. 153-159, 1991.
[8] P. Godefroid and P. Wolper, "A Partial Approach to Model Checking," *Information and Computation*, vol. 110, no. 2, pp. 305-326, May 1994.
[9] *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*, M.J.C. Gordon and T.F. Melham, eds., Cambridge Univ. Press, 1993.
[10] M.G. Gouda, N.J. Multari, "Stabilizing Communication Protocols," *IEEE Trans. Computers*, vol. 40, no. 4, Apr. 1991.
[11] J.-H. Hoepman, "Uniform Determination Self-Stabilizing Ring-Orientation on Odd-Length Rings," *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, pp. 265-279, 1994.
[12] G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279-295, May 1997.
[13] S.-T. Huang, "Leader Election in Uniform Rings," *ACM. Trans. Programming Language and Systems*, vol. 15, no. 3, pp. 563-573, July 1993.
[14] A. Israeli and M. Jalfon, "Uniform Self-Stabilizing Ring Orientation," *Information and Computation*, vol. 104, pp. 175-196, 1993.
[15] S.S. Kulkarni, J. Rushby, and N. Shankar, "A Case-Study in Component-Based Mechanical Verification of Fault-Tolerant Programs," *Proc. Workshop on Self-Stabilization (WSS '99)*, pp. 33-40, 1999.
[16] K. L. McMillan, *Symbolic Model Checking*. Kluwer Academic, 1993.
[17] S. Owre, J.M. Rushby, and N. Shankar, "PVS: A Prototype Verification System," *Proc. 11th Int'l Conf. Automated Deduction (CADE-11)*, pp. 748-752, 1992.
[18] A. Pnueli, "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. Foundation of Computer Science*, pp. 46-57, 1977.
[19] I.S.W.B. Prasetya, "Mechanically Verified Self-Stabilizing Hierarchical Algorithms," *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, pp. 399-415, Apr. 1997.
[20] S. Qadeer and N. Shankar, "Verifying a Self-Stabilizing Mutual Exclusion Algorithm," *Proc. IFIP Working Conf. Programming Concept and Methods (PROCOMET '98)*, pp. 424-443, June 1998.
[21] S. Rajan, N. Shankar, and M.K. Srivas, "An Integration of Model Checking with Automated Proof Checking," *Proc. Seventh Workshop on Computer-Aided Verification (CAV '95)*, pp. 84-97, 1995.
[22] M. Schneider, "Self-Stabilization," *ACM Computing Surveys*, vol. 25, no. 1, Mar. 1993.
[23] S.K. Shukla, D.J. Rosenkrantz, and S.S. Ravi, "Simulation and Validation of Self-Stabilizing Protocols," *Proc. SPIN '96 Workshop*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32, 1997.
[24] J.M. Spinelli, "Self-Stabilizing Sliding Window ARQ Protocols," *IEEE/ACM Trans. Networking*, vol. 5, no. 2, pp. 245-254, 1997.
[25] O. Theel and F.C. Gärtner, "An Exercise in Proving Convergence through Transfer Functions," *Proc. Fourth Workshop on Self-Stabilizing Systems*, pp. 41-47, 1999.
[26] N. Umemoto, H. Kakugawa, and M. Yamashita, "A Self-Stabilizing Ring Orientation Algorithm with a Smaller Number of Processor States," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 6, June 1998.
[27] A. Valmari, "A Stubborn Attack on State Explosion," *Proc. Second Workshop on Computer Aided Verification (CAV '90)*, pp. 156-165, June 1990.

Tatsuhiro Tsuchiya received the ME and PhD degrees in computer engineering from Osaka University in 1995 and 1998, respectively. He is currently an assistant professor in the Department of Informatics and Mathematical Science at Osaka University. His research interests are in the areas of distributed computing and formal verification. He is a member of the IEEE.

Shin'ichi Nagano received the ME and PhD degrees in computer engineering from Osaka University in 1996 and 1999, respectively. In 1999, he joined Toshiba Corporate Research and Development Center, where he is currently a research staff member. His research interests include multiagent systems and formal verification of communication protocols. He is a member of the IEEE.

Rohayu Bt Paidi received the BE in computer engineering from Osaka University in 1999. She is currently with Matsushita Electric Company, Malaysia. She is a member of the IEEE.

Tohru Kikuno the received MS and PhD degrees from Osaka University in 1972 and 1975, respectively. He joined Hiroshima University from 1975 to 1987. Since 1990, he has been a professor in the Department of Informatics and Mathematical Science at Osaka University. His research interests include the quantitative evaluation of software development processes, the analysis and design of fault-tolerant systems, and the design of procedures for testing communication protocols. He served as a program cochair of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98) and the Fifth International Conference on Real-Time Computing Systems and Applications (RTCSA '98). He is a member of the IEEE.